Predicating Load Latencies Using Cache Profiling

Santosh G. Abraham, B. Ramakrishna Rau
Compiler and Architecture Research

HPL-94-110
November, 1996

Predicting Load Latencies Using Cache Profiling

Santosh G. Abraham, B. Ramakrishna Rau
Compiler and Architecture Research
HPL-94-110
November, 1994

cache profiling, simulation, cache management instructions, prefetching, memory system analysis, program characterization

Due to increasing cache-miss latencies, cache control instructions are being implemented for future systems. In order to investigate the potential benefit of using these instructions in compiling a broad range of applications, we study the memory referencing behavior of individual machine-level instructions using simulations of fully-associative caches. Our objective is to obtain a deeper understanding of useful program behavior that can be eventually employed at optimizing programs and to motivate architectural features aimed at improving the efficacy of memory hierarchies. Our simulation results show that a small number of load instructions account for a majority of data cache misses in the SPEC89 benchmarks. Most load instructions rarely miss the data cache and a few instructions have miss rates much higher than the global miss ratio. We develop a model where the cost associated with scheduling loads with miss latencies is parameterized. Using this model, we explore the potential net benefit associated with profile-based scheduling of load instructions where some loads are selected for scheduling with miss latencies based on their behavior under profiling. Finally, we investigate the sensitivity of cache profiling information to input data sets. In many cases, the profile information obtained using one data set is quite effective in predicting the behavior of a program on other data sets, but in some cases the predicted caching behavior is quite different from the actual behavior.

1 Introduction

Processor performance has been increasing at 50% per year but memory access times have been improving at 5-10% per year only. As a result, the latency of cache misses in processor cycles is increasing rapidly. Besides the increasing cache-miss latencies, another trend is towards processors that issue a larger number of instructions/operations per cycle, such as VLIW or superscalar processors. In these systems, the effective cache-miss penalty is proportional to issue width. Therefore, it is important to develop techniques to reduce the cache miss ratio and tolerate the latency of cache misses.

Current compilers use *all-hit scheduling* where all loads are scheduled using the cache-hit latency. Even though a lockup-free cache prevents a stall on a cache miss, the processor often stalls soon afterwards on an instruction that requires the data that missed the cache. The dynamic scheduling capabilities needed to cope with the highly variable latencies associated with cache misses are expensive and could degrade the processor cycle time. The alternative of scheduling all loads with the cache-miss latency (referred to subsequently as *all-miss scheduling)* is not feasible for most programs because it requires considerable instruction level parallelism (which may not be present) if it is to be successful and increases register pressure. Studies investigating load delay slots in RISC processors support this observation. Scheduling with the miss latency eliminates a large portion of the value of having a cache, viz., the short average latency, and retains only the reduction in main memory traffic. In this report, we investigate the potential improvements of a third alternative, *selective load scheduling* where some loads are selected for scheduling with the cache-miss latency while other loads are scheduled with the cache-hit latency.

In order to effectively support selective load scheduling, systems should minimally have a lockup-free cache and preferably instructions that permit the software to manage the cache, e.g., DEC Alpha [1]. Kathail *et al* [2] describe the HPL PlayDoh architecture which supports a comprehensive set of load/store operation specifiers for managing the data cache. We review the set of *source-cache* or *latency* specifiers for loads provided in the HPL PlayDoh architecture that specifies the highest level in the memory hierarchy where the data is likely to be found. After an earlier phase of compilation has inserted the latency specifiers, the instruction scheduler preferentially schedules *long-latency* loads which tend to miss the cache and all other *short-latency* loads with the cache-hit latency. In a VLIW processor, the architectural latency of a load is determined using its latency specifier. If the behavior of loads is consistent with their latency specification, a major source of non-deterministic high-variance latencies which greatly impacts the performance of VLIW processors is eliminated. Superscalar processors can also use the latency specifiers for dynamic scheduling. We also review a set of *target cache* or *level-direction* specifiers for load/store instructions that install data at specific levels in the memory hierarchy [2]. Through these instructions, the software can manage the contents of caches at various levels in the memory hierarchy and potentially improve cache-hit ratios and/or improve the accuracy of latency specification.

In selective load scheduling, the compiler must tag each load with a latency specifier. For regular scientific programs, compile-time analysis can identify the loads to schedule with the cache-miss latency. But, for irregular programs such as `gcc` or `spice`, compile-time techniques have not been developed. A key issue that will determine the effectiveness of latency specification is the predictability of load latencies for such irregular programs. In this report, we study the behavior of individual load/store instructions with respect to their data cache accessing behavior using cache simulations. Unlike earlier research that reported overall miss ratios, we determine the miss ratios of individual load/store instructions. Though a relatively small number of loads account for a large majority of memory references, a much smaller number of loads account for most cache misses. We show that a large fraction of memory references are made by instructions that miss extremely infrequently. We use the profiling information obtained by these cache simulations to label loads as short or long latency instructions. In addition to the results that were presented earlier in Abraham *et al* [3], we present a model where we characterize the net benefit of selective load scheduling as a function of the scheduling costs associated with scheduling loads with miss latencies. Furthermore, we investigate the sensitivity of cache profiling to different input data sets.

There are some important factors that make the usage of load profiling information more difficult than the more established branch profiling. Firstly, load profiling is inherently more expensive than branch profiling by one or two orders of magnitude. However, there are several methods to reduce this cost, such as using trace sampling and inline simulation of cache hits. Secondly, unlike branches whose targets are usually known, the address of a load is often not known sufficiently in advance. If the address is not known, the long latency load cannot be moved ahead in the schedule sufficiently. Thus, if there is little flexibility in scheduling loads with miss latencies, then selective load scheduling can only be marginally better than conventional all-hit scheduling. Finally, the results in this report indicate that cache profiling is more sensitive to changes in input data set than branch profiling. Further work is needed to develop more sophisticated profiling techniques that are less sensitive to changes in data set. Even if cache profiling is not feasible for production compiler use, the cache profiling results may lead to compiler techniques that generate similar information.

2 Related Work

Several researchers have investigated prefetching to reduce cache miss stalls. Callahan and Porterfield [4] investigate the data cache behavior of scientific code using source-level instrumentation. For a suite of regular scientific applications, they show that a few source-level references contribute to most of the misses. In a later paper, Callahan and Porterfield [5] present techniques for prefetching to tolerate the latency of instructions that miss. They investigate both complete prefetching where all array accesses are prefetched, and selective prefetching where the compiler decides, based on the size of the cache, what data to prefetch. McNiven and Davidson [6] evaluate the reductions in memory traffic obtainable through compiler-directed cache management. Klaiber and Levy [7] and Mowry *et al* [8] also investigate selective prefetching using compiler techniques for scientific code. Mowry *et al* [8] combine prefetching with software pipelining and further investigate the interaction between blocking and prefetching. Chen and Baer [9] investigate hardware-based prefetching and moving loads within basic blocks statically to tolerate cache-miss latencies for some SPEC benchmarks. Chen *et al* [10] reduce the effect of primary cache latency by *preloading* all memory accesses for the SPEC benchmarks.

In this report, we describe a framework for software cache control that subsumes prefetching. We illustrate the performance improvements obtainable through software cache control using the matrix multiply program. Unlike Callahan and Porterfield [4], we investigate the caching behavior of *machine-level* instructions for irregular floating-point programs such as `spice` and integer programs such as `gcc`. We also evaluate the costs and benefits of prefetching using load/store profiling.

Profile guided optimizations play an important role in code optimization. Branch profiling has been used in scheduling multiple basic blocks together to increase instruction level parallelism in VLIW and superscalar machines by Fisher [11] and Hwu *et al* [12]. Fisher and Freudenberger [13] present a new measure for evaluating branch predictability and demonstrate that branch profiling is largely insensitive to a range of different input sets for the SPEC benchmarks. Krall [14] demonstrates that branch profiling can be improved by using correlations between the behavior of branches and replicating branches which have a few distinct patterns of behavior. Profiling has also been used in instruction cache management by McFarling [15] and by Hwu and Chang [16], register allocation by Chow and Hennessy [17], and inlining by Chang *et al* [18]. This work proposes using load/store instruction profiling for instruction scheduling and cache management. MemSpy [19] is a profiling tool that can be used to identify source-level code segments and data structures with poor memory system performance. Our work focuses on opportunities for optimization of cache performance through profiling of machine-level load/store instructions.

The rest of the report is structured as follows. Section 3 develops the machine model. Section 4 illustrates several concepts underlying our approach using a matrix multiply program. Section 5 describes the profiling experiments and the results. Section 6 presents the results on profiling with multiple input data sets. Section 7 is the conclusion and future work.

3 Instruction Extensions for Software Cache control

In this section, we review a comprehensive, orthogonal set of modifiers to loads/stores originally described by Kathail *et al* [2] that can be used to specify the latency of loads as well as to control the placement of data in the memory hierarchy.

In most systems, caches are entirely managed by the hardware. But in order to control the placement of data in hardware managed caches, the compiler [1] must second-guess the hardware. Furthermore, since the cache management decisions are not directly evident to the software, the load/store latencies are non-deterministic. At the other extreme, local or secondary memory is entirely managed by the program. The main disadvantage of an explicitly managed local memory is that the software is forced to make all management decisions.

We describe a mixed implicit-explicit managed cache that captures most of the advantages of the two extremes. A *mixed cache* provides software mechanisms to explicitly control the cache as well as simple default hardware policies. The software mechanisms are utilized when the compiler has sufficient knowledge of the program's memory accessing behavior and when significant performance improvements are obtainable through software control. Otherwise, the cache is managed using the default hardware policies.

We now describe the load/store instructions in the HPL PlayDoh architecture [2] in the remainder of this section [2]. In addition to the standard load/store operations, the PlayDoh architecture provides explicit control over the memory hierarchy and supports prefetching of data to any level in the hierarchy.

The PlayDoh memory system consists of the following: main memory, second-level cache, first-level cache and a first-level data prefetch cache. The data prefetch cache is used to prefetch large amounts of data having little or no temporal locality without disturbing the conventional first-level cache. The first- and second-level caches may either be a combined cache or be split into separate instruction and data caches. This memory hierarchy can be extended to several levels. For the purposes of this report, we are primarily interested in managing the first-level cache.

There are two modifiers associated with each load operation which are used to specify the load latency and to control the movement of data up and down the cache hierarchy. The first modifier, the latency and source cache specifier, serves two purposes. First, it is used to specify the load latency assumed by the compiler. Second, it is used by the compiler to indicate its view of where the data is likely to be found. The second modifier, the target cache specifier, is used by the compiler to indicate its view

1. In this section, when we say compiler we mean programmer/compiler.
2. HPL PlayDoh is a parametric architecture that has been defined to support research in instruction-level parallel computing.

of the highest level in the cache hierarchy to which the loaded data should be promoted for use by subsequent memory operations. (Note that the first-level cache is the highest level in the hierarchy.) The second specifier is used by the compiler to control the cache contents and manage cache replacement. Each of these specifiers can be one of the following: *V* for data prefetch cache, *C1* for first-level data cache, *C2* for second-level data cache, *C3* for main memory.

Non-binding loads cause the specified promotion of data in the memory hierarchy without altering the state of the register file. In a non-binding load, the destination of the load is specified as register 0. For convenience, we will use the terms *pretouch* and *prefetch* to refer to non-binding loads that specify the target cache as *C1* and *V* respectively. In contrast to regular loads, prefetches and pretouches bring the data closer to the processor without tying up registers and thereby increasing register pressure. Also, *long-latency* loads are binding loads that specify *C2* as the source cache. Other binding loads are *short-latency* loads.

A store operation has only a target cache specifier. As with the load operations, it is used by the compiler to specify the highest level in the cache hierarchy at which the stored data should be installed to be available for use by subsequent memory operations. In the rest of this report, we limit the discussion to a two-level memory hierarchy, consisting of a cache, *C* and a memory, *M*.

In a *latency-stalled* machine, there is no interlock or check for uses of the load. Instead, each operation specifies its latency and if the result is not delivered within the specified latency the entire processor stalls till the result is available. Since loads have widely differing latencies (depending on the level of the memory hierarchy at which the data is found), the specification of a latency consistent with the actual latency has important performance implications. If a short-latency load consistently misses the cache, then each such miss stalls the processor till the data is obtained from some deeper level of the memory hierarchy. If a long-latency load consistently hits the cache, then the compiler has extended the load to use distance at some potential increase in schedule length and lifetimes, without a balancing gain in reduced stall cycles. In an interlocked machine, where the processor stalls only on the use of a result, the processor itself may not use the specification of load latencies. However, the compiler may internally associate distinct latencies with operations to suitably schedule load operations so that memory stall cycles associated with missing load operations are reduced.

4 Programmatic control of memory hierarchy: An example

Although our interest extends beyond regular programs, we illustrate the concepts underlying this report using the familiar blocked matrix multiply example. The usefulness of the cache control instructions can be demonstrated concisely and effectively on the matrix multiply example. Furthermore, the optimized blocked matrix multiply with the cache control instructions exhibit ideal behavior in the sense that a few instructions consistently miss the cache and other instructions always hit the cache. This information can be exploited to improve program performance by choosing the right load instruction from the repertoire described in the previous section. In the experimental results section, the other benchmarks can be better understood by contrasting their behavior with that of blocked matrix multiply.

4.1 LRU statistics for blocked matrix multiply

For reasonable cache sizes, the regular unblocked matrix multiply incurs approximately $2n^3$ cache misses out of a total of $3n^3$ data references. Figure 1 shows a blocked version of the matrix multiply program with a block size of m=32. In general, the block size m in the program is chosen so that $m^2 + 3m \leq$ cache size. Satisfying this inequality ensures that a square block of size $m \times m$ of B can be kept resident in cache throughout its use. Compared to the regular matrix multiply, the misses due to re-references to elements in the B array are eliminated and the overall number of misses is reduced to approximately $2n^3/m$.

```
      PROGRAM matmul
C     constraint: n has to be an integer multiple of
C     the block size m
      parameter (n = 128, m = 32)
      real A(n, n), B(n, n), C(n, n)
      do 10 jj = 1, n, m
         do 10 kk = 1, n, m
            do 10 i = 1, n
               do 10 j = jj, jj+m-1
                  do 10 k = kk, kk+m-1
                     C(i, j) = C(i, j) + A(i, k)*B(k, j)
 10   continue
```

Figure 1.Blocked matrix multiply

Blocking has been widely used by programmers for regular scientific applications and has recently been incorporated as an automatic transformation in some research and commercial compilers. Our objective is to illustrate the regularity of memory references of load/store instructions and this objective can be realized using either the regular or blocked matrix multiply. Since the validity of new optimization approaches is best illustrated on programs that have already been optimized using existing techniques, we choose the blocked version of matrix multiply for further consideration.

Table 1 shows the LRU stack distributions for the blocked matrix multiply. Each entry in the LRU (Least Recently Used) stack is a data address. Each entry, e_j at a depth j, occupies a position in the LRU stack below all other entries that have been referenced after e_j. In a fully-associative LRU data cache of size C lines, references that access entries below C in the LRU stack are misses and the rest are hits. The LRU stack distribution shows the number of times a static program access referenced a data item within a range of depths in the LRU stack. Besides showing the actual LRU stack distribution when n=128, m=32, Table 1 also shows a general expression for the stack depth distribution and the values of the loop indices in the program for which these stack depths are encountered. Of the n^3 references made by each static source-level reference, n^2 are the initial references to the array of size n^2. These initial references are characterized as compulsory misses. The remaining n^3-n^2 references are made at different depths for different references.

4.2 Multi-issue machine model

In the remaining subsections of this section, we illustrate the performance improvements obtainable through the use of level-directed, latency-specifying loads/stores on a blocked matrix multiply. In order to quantify the performance improvements, we use the following multi-issue machine model. Except for this section, the rest of the report is independent of the following machine model.

Our multi-issue model issues a total of seven operations per cycle: two memory loads/stores, two address computations, one add, one multiply, and one branch. Table 2 lists the types of operations and their latencies. The latency of loads is 1 if it is in the first-level cache or buffer and 20 otherwise.

4.3 No Data Cache

Now, we examine the performance of the blocked program in Figure 1 on a range of architectural alternatives. Table 3 shows four metrics of interest for five different architectural alternatives. The ideal execution time is the execution time on our sample machine ignoring loop and memory overheads of a matrix multiply code scheduled either with a load latency of 1 or 20 cycles. The cache miss penalty is the number of cycles spent stalling for cache misses. The actual execution time is the sum of the first two times. The memory traffic is the total number of non-cached accesses. In each case, we present a general expression and a number for the particular choice of n=128 and m=32.

First, consider a machine without a data cache representative of VLIW machines such as the Cydrome Cydra 5 [20, 21] or Multiflow TRACE family [22]. The ideal execution time assumes a schedule with a memory latency of 20 cycles. Since there is no cache, the cache miss penalty is zero and every reference is satisfied by main memory. This example illustrates that in regular scientific applications there is usually sufficient parallelism to schedule loads to hide the memory latency. However, one memory access is required per floating-point operation and this level of memory traffic can be sustained only by expensive memory systems.

Table 1. LRU stack depth distributions for blocked matrix multiply

	A(i,k)	B(k,j)	C(i,j)	C(i,j)
Depth	Load	Load	Load	Store
1-16	0	0	2.03M ($n^3*(m-1)/m$ at depth 3) $[k \neq kk]$	2.1M (n^3 at depth 1)
17-64	0	0	0	0
65-1K	2.03M ($n^3*(m-1)/m$ at depth $2m+2$) $[j \neq jj]$	0	0	0
1K-4K	0	2.1M (n^3-n^2 at depth m^2+3m) $[i \neq 1]$	0	0
4K-16K	0	0	49,152 ($n^2*(n/m-1)$ at depth $2m^2+2nm$) $[k \neq 1\ k = kk]$,	0
16K-64K	49,152 ($n^2*(n/m-1)$ at depth $(n^2+3nm+m^2)$) $[j \neq 1, j = jj]$	0	0	0
Compulsory misses	16,384 n^2 misses $[j = 1]$	16,384 n^2 misses $[i = 1]$	16,384 n^2 misses $[k = 1]$	0

4.4 Conventional data cache

Secondly, consider a machine with a first-level data cache which is larger than $2n + 1$ and $m^2 + 3m$ words and is smaller than $n^2 + 2n$. The cache stalls the processor on a cache miss till the miss is serviced and the cache-miss latency is 20 cycles. Loads are scheduled by the compiler with the cache-hit latency. Most current RISC systems have a cache organization and scheduling approach similar to this machine. Com-

Table 2. Instruction latencies

Pipeline	Number	Operations	Latency
Memory port	2	Load	1/20
		Store	1
Address ALU	2	Address add/subtract	1
Adder	1	Floating-point add/ subtract	2
		Integer add/subtract	1
Multiplier	1	Floating-point multiply	4
		Integer multiply	2
Instruction	1	Branch	2

pared to the earlier alternative, the combination of caching and blocking transformations reduces memory traffic by a factor of m. This machine has an ideal execution time that is lower and has a significantly less expensive memory system. However, the overall execution time is significantly degraded by the cache miss penalty.

4.5 Lockup-free cache with out-of-order execution

Consider a lockup-free cache in conjunction with a dynamically-scheduled processor that supports out-of-order execution and is similar to our model processor in issuing capabilities. Assume that the blocked matrix-multiply is scheduled for such a machine with a nominal load latency of one. Assuming the system is effective at hiding the cache miss latency, the cache miss penalty is zero. Compared to the "no data cache" system, the lockup-free cache system has greatly reduced memory traffic requirements and compared to the previous cache alternative, the current system appears to have better overall execution times. But, in order to hide the cache miss la-

Table 3. Performance metrics on architectural alternatives

	Ideal execution time	Cache miss penalty	Actual exec. time	Memory traffic
No data cache	$n^3\frac{m+26}{m}$ 3.80M	-	$n^3\frac{m+26}{m}$ 3.80M	$2n^3+\frac{n^3}{m}$ 4.26M
Data cache	$n^3\frac{m+7}{m}$ 2.56M	~ $40\frac{n^3}{m}$ 2.62M	~ $n^3\frac{m+47}{m}$ 5.18M	$\frac{2n^3}{m}+n^2$ 0.147M
Lockup-free cache + dyn. schedul-ing	$n^3\frac{m+7}{m}$ 2.56M	-	$n^3\frac{m+7}{m}$ 2.56M	$\frac{2n^3}{m}+n^2$ 0.147M
Labeled loads	~ $n^3\frac{m+8}{m}$ 2.60M	-	~ $n^3\frac{m+8}{m}$ 2.60M	$\frac{2n^3}{m}+n^2$ 0.147M
Ideal data cache	$n^3\frac{m+7}{m}$ 2.56M	-	$n^3\frac{m+7}{m}$ 2.56M	$3n^2$ 0.049M

tency, the hardware must be capable of dynamically scheduling from a window of 140 instructions (7 instrs./cycle × 20 cycles) which may impact the processor cycle time. Though dynamically-scheduled systems with lockup-free caches have the potential for eliminating the cache miss penalty, the large scale schedule reorganization required for overlapping the relatively high cache miss latency makes their practicality somewhat suspect.

4.6 Conventional system scheduled using miss latencies

A solution to the above problem is to statically-schedule the code with a nominal load latency equal to the cache-miss latency instead of the cache-hit latency. This solution is effective for the matrix multiply example but is not likely to be as effective in general. For programs with a modest amount of instruction level parallelism, scheduling loads with the cache-miss latency increases the critical path, creates empty slots in the schedule and reduces overall performance.

4.7 Labeled loads/stores

We consider the use of level-directed, latency-specifying loads/stores that were described in Section 3. For each load instruction, the latency specifier is determined through compile-time analysis (possible for dense matrix computations) or through cache-simulation based profiling. Assuming that each source program access is translated into one machine-level load instruction, there are three loads, one each for the arrays A, B, C. Since the miss ratio for all these loads is fairly low: 0.031, 0.008, and 0.031 respectively for A, B, C, any general scheme would label all these loads as short latency loads and schedule the entire program using cache-hit latencies. Thus, there is no difference between this scheme and the "Conventional data cache" scheme.

Though it is clear from Table 1 that each load instruction has two or three distinct cache behavior, the simple labeling scheme above does not have sufficient resolution to discriminate between these behavioral patterns.

4.8 Labeled loads/stores after peeling

Loop peeling is a simple compiler transformation where a specific iteration of the loop is "peeled off" from the rest of the loop and replicated separately. In many situations, the accesses in the first iteration bring data into the cache, tending to miss, and the last iteration references data for the last time without subsequent useful reuse. Therefore, peeling the first and last iterations of the loop, enables the identification of loads/stores with a high miss rate or without reuse.

Consider peeling the first iterations of the *i* and *j* loop from the program in Figure 1, so that the loop body is replicated a total of four times. Table 4 gives the miss ratio (when n=128, m=32) for the four distinct accesses to each of the arrays in the peeled code. From Table 4, each of the references in the peeled program have very regular caching behavior, i.e., either a reference always hits or always misses in an LRU fully-associative cache. Each reference in the program is annotated with the number of times it hits or misses and the corresponding depths in the LRU stack. If the reference always hits, its latency is specified as short latency; otherwise, it is labeled as a long latency reference.

Also, if a reference brings in data that is not reused before being replaced from the LRU cache, the data is bypassed and not installed in cache. Though there is reuse on data brought in by references to C, this reuse occurs at a large depth in the LRU stack and therefore all references to C miss the cache. Therefore, data brought in by C is not installed and is bypassed. Bypassing the cache prevents the data from polluting the cache by displacing other useful data. In this example, data brought in by a reference to A and B is generally reused and therefore all references to A and B are directed to the first-level cache.

Table 4. Misses of individual loads/stores in peeled program

i	j	Description	A	B	C
1	jj	Compulsory misses	n	n^2/m	n/m
		Other misses	$n(n/m-1)$	0	$(n/m)(n/m-1)$
		Hits	0	0	0
		Overall Miss ratio	1.0	1.0	1.0
		Short/long latency	Long	Long	Long
		Cache install	Yes	Yes	No
1	!= jj	Compulsory misses	0	$\frac{(n^2 \cdot (m-1)}{m}$	$(n/m) \cdot (m-1)$
		Other misses	0	0	$(n/m) \cdot ((n/m)-1) \cdot (m-1)$
		Hits	$\frac{(n^2 \cdot (m-1))}{m}$	0	0
		Overall Miss ratio	0.0	1.0	1.0
		Short/long latency	Short	Long	Long
		Cache install	Yes	Yes	No
!=1	jj	Compulsory misses	$n \cdot (n-1)$	0	$(n/m) \cdot (n-1)$
		Other misses	$n \cdot (n-1) \cdot ((n/m)-1)$	0	n
		Hits	0	$\frac{n^2 \cdot (n-1)}{m}$	$(n-1) \cdot ((n/m)-1)$
		Overall Miss ratio	1.0	0	1.0
		Short/long latency	Long	Short	Long
		Cache install	Yes	Yes	Yes
!=1	!= jj	Compulsory misses	0	0	$(n/m) \cdot (n-1) \cdot (m-1)$
		Other misses	0	0	$(n/m) \cdot ((n/m)-1) \cdot (n-1) \cdot (m-1)$
		Hits	$\frac{n^2(n-1)(m-1)}{m}$	$\frac{n^2(n-1)(m-1)}{m}$	0
		Overall Miss ratio	0.0	0.0	1.0
		Short/long latency	Short	Short	Long
		Cache install	Yes	Yes	No

The compiler analysis required for the blocking transformation determines that the i=1 and j=1 iterations bring data initially into the cache and that the reuse of data occurs on $i \neq 1$ and $j \neq 1$ iterations for the arrays B and A respectively. This analysis will therefore suggest peeling to clone the accesses to these arrays.

Just as peeling enabled us to isolate references that miss in cache, peeling can also be used to identify references on which there is no reuse. Generally, the first iteration brings in data and the last iteration accesses data without any subsequent useful reuse. In our running example, there is no reuse of data accessed in the last iterations of the i and j loops by the arrays B and A respectively. Therefore, once the last iterations of i and j are peeled off, the references that have no reuse are exposed. These A and B references will be labeled short latency but will be directed to a lower level of the cache.

The above discussion demonstrates that compiler transformations such as peeling can be useful prior to profiling in order to improve the resolution of the profiling. Similarly, loop unrolling by a factor equal to the cache line size is useful in identifying the references that miss in a stride-one access pattern. In practice, peeling and loop unrolling prior to profiling are not necessary. Instead, profiling information on each load is gathered at a finer level of granularity that is trajectory sensitive. More precisely, a range of bins are maintained for accumulating hit-miss information for each static load access. The actual bin that is chosen depends on the sequence of basic blocks through which control arrives at a load. For instance, two bins can be maintained for each load, one that is updated when the control reaches a load in a loop for the first time and the other that is updated when the loop iterates. The techniques employed may be similar to those used to improve static branch profiling results [14].

In the case of regular scientific programs such as matrix multiply, extensions to current compiler technology are sufficient to characterize loads as short or long latency and to label them as cache install or bypass. Current compiler techniques are unlikely to be successful at characterizing load instructions in a more general class of programs. We propose the use of cache profiling to study the regularity of the memory referencing behavior of load instructions in such programs. We study the effectiveness of using this profiling information to specify the latency of load instructions. Our objective is not to advocate a specific scheme such as a profile-driven scheme for labeling loads but is targeted rather at developing a deeper understanding of useful program behavior that can be eventually employed at optimizing programs and at motivating architectural features aimed at improving the efficacy of memory hierarchies. For instance, the profiling experiments in the next section may motivate the development of compiling techniques that can identify latency and reuse of memory references in a broader class of programs. These compiling techniques may in turn be superior to profile-driven labeling of loads for a broad class of programs in certain situations.

5 Simulation Experiments

In this section, we characterize the cache behavior of loads by performing cache simulations. Our goal is limited to evaluating the potential benefit of profile-driven source-cache specification of loads; we do not discuss target-cache specification further in this report.[1] We first describe the simulation environment and the benchmarks. We investigate the regularity of the caching behavior of loads both with respect to the distribution in misses among loads and the distribution of miss ratios. We statically annotate the load/store instructions with prefetches using the profiling information and present two performance metrics related to the cost and benefit of profile based annotation.

5.1 Environment

We use four-word line size fully-associative 16KB caches in our program characterization, because the results are obtained at a higher level of abstraction without the artifacts introduced by limited associativity. We have confirmed that the general nature of the results apply for two-way set-associative caches [23]. We simulated the caches either for 100 million addresses or till completion of the program. We use the stack processing technique of Mattson *et al* [24] to simulate fully-associative caches under LRU and MIN replacements and we use tree implementations of the stack for efficiency. The stack represents the state of a range of cache sizes, with the top C lines being contained in a cache of size C lines. The initial program characterization in Section 5.3 is based on normal LRU replacement because it is commonly used in practical caches. In the remainder of the section, the objective is to do better than hardware schemes by utilizing future program accessing information. Therefore, we use MIN in the profiling step to capture future program behavior and we then use software control and default LRU replacement to measure performance improvements. A cache of size C using the MIN replacement scheme prefetches data into the cache to always maintain the next C distinct lines that will be referenced. Since the depth at which a reference is inserted in the MIN stack is the depth at which it is rereferenced in the LRU stack, the distribution of insertion depths in the MIN stack is actually obtained using a modified LRU stack simulator.

We define some relevant terms and measures in the following. The *global miss ratio* is the fraction of references that miss the data cache over the entire program. The *miss ratio* of a single static load is the number of times the references generated by a load misses over the total number of references generated by that load. The *miss fraction* of a load is the number of misses generated by that load over the total number of misses in the program. Similarly, the *reference fraction* of a load is the number of references generated by that load over the total references generated by the entire pro-

1. Cache replacement can be managed using the target cache specifiers described in Section 2 to reduce traffic between levels of the memory hierarchy. Initial results on using profiling to manage cache contents show a reduction in traffic of less than 10% [23].

gram. The *cumulative miss fraction* is the sum of the miss fraction of a set of loads and similarly, the *cumulative reference fraction* is the sum of the reference fractions of a set of loads. The *load fraction* is the number of dynamic load operations executed over total number of operations executed in the program.

5.2 Benchmarks

We present results on a limited number of programs consisting of seven SPEC89 benchmarks and `matrix`, which is the peeled blocked matrix multiply example that was described in detail in the previous section. In this section, we present detailed results on four benchmarks for our experiments: `matrix`, `dnasa`, `tomcatv` and `eqn`. In the next section, we will present similiar results on four other SPEC89 benchmarks: `gcc`, `xlisp`, `espresso`, and `spice`. For the second set of benchmarks, multiple data sets were available and we present the results of profiling with one data set and measuring the results on another distinct data set.

5.3 Distribution of references and misses

In this subsection, we investigate the distribution of references and misses among the loads in a program. It is widely known that, in most programs, a relatively small part of the program accounts for most of the execution time. Consequently, we expect that most references are due to a few loads in most programs. We examine whether the misses in a program are also concentrated among a few static loads.

We sort the loads in a program by decreasing reference ratios and form a list of all loads with the load responsible for most references at the head of the list. The graphs in Figure 2 plot the cumulative reference fraction of a set of load instructions on the y-axis and the number of loads in the set on the x-axis. The x-axis is on a log scale in order to magnify the left hand region showing the behavior of instructions with high reference coverage. We also form a separate list of loads sorted by decreasing miss ratios. In the second curve, we plot the cumulative miss fraction of a set of load instructions versus the number of loads in that set. Now, the loads are ordered with the load accounting for the most misses at the head of the list. For `eqn`, 10 instructions account for almost 90% of the dynamic references and a possibly different set of 10 instructions accounts for 94% of all misses.

For programs such as `tomcatv`, 100 instructions account for almost all the misses but the 100 most frequently referenced instructions only account for 70% of the references. The fact that a certain number of instructions account for a much larger fraction of cache misses than overall references demonstrates that the concentration of cache misses among a few instructions is not merely due to a concentration of references among a few instructions. Because a few instructions account for most cache misses, it is more likely that manual or compiler intervention techniques can be developed to handle cache misses.

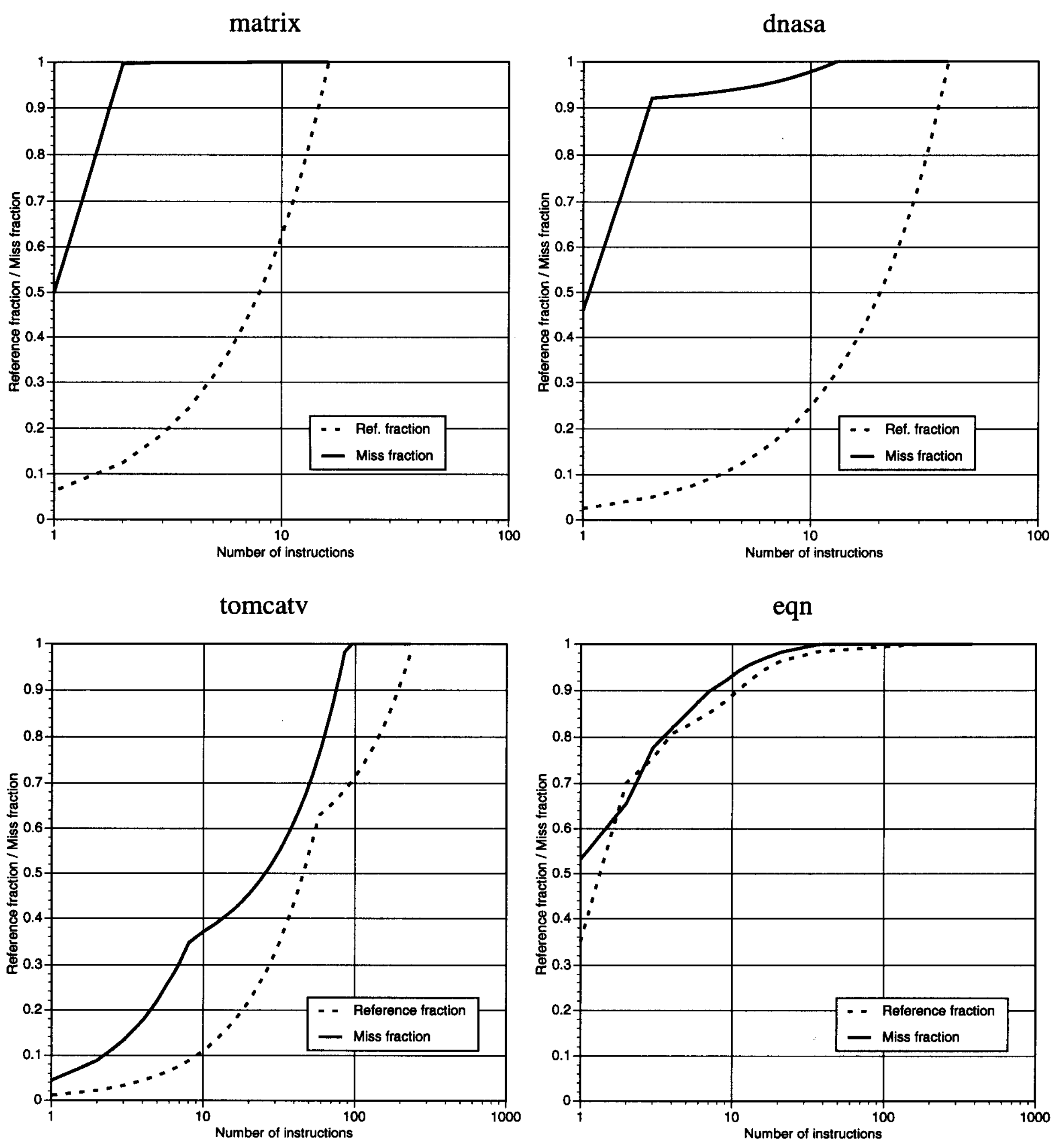

Figure 2.Cumulative reference and miss fractions vs. sorted loads

5.4 Distribution of dynamic trace annotations

In this subsection, we consider annotating each reference using fully-associative cache simulation under MIN. For a cache with C lines, a reference is annotated with 'was replaced' if it is the first reference to access a cache line after it has crossed a depth of C in the MIN stack, i.e. after it has been replaced; otherwise, it is annotated with 'was retained'. We examine the distribution of annotations on references made by individual instructions. For each static instruction, we maintain a count of the number of times it accesses references annotated with 'was replaced', i.e. count of misses per instruction.

Branch profiling is useful because certain branches tend to be taken and others tend to be not taken and profiling classifies branches into these two types. Each branch type can be optimized for its predicted behavior. In this section, we examine a similar issue with respect to the cache behavior of loads using the annotation counts per instruction. We can classify loads into two categories through profiling based on cache simulation: *hit-instructions* that tend to hit and *miss-instructions* that tend to miss while accessing the data cache. As in branch profiling, the value of such a classification depends on how closely the observed behavior matches the predicted behavior.

To determine how often the observed behavior is different from the predicted behavior, we sort the individual loads by their miss ratios. In Figure 3, the cumulative reference fraction on the x-axis is the fraction of the total references made by some number of instructions starting from the head of the sorted list. Again, the x-axis is on a log scale to magnify the interesting regions of the four graphs. The y-axis shows the miss ratio of the last instruction removed from the sorted list to reach a certain cumulative reference fraction. The shape of the curves for `matrix` and `dnasa` suggests that the miss ratio of instructions is either large or close to zero. For the three programs, `matrix`, `dnasa`, and `tomcatv`, a large fraction of references are accounted for by instructions with a miss ratio of zero and relatively few references are accounted for by instructions in the transitional region. The exception among these four programs is `eqn` where almost all references are made by instructions with a non-zero but small miss ratio.

Though the instructions that hit tend to have a hit ratio close to one, the instructions that miss do not tend to have a miss ratio of one. Thus, the predictability of miss-instructions is not as good as hit-instructions. Firstly, there are many more hits than misses globally and secondly, our cache model assumes a line size of four words. Unit-stride reference streams that always miss in a cache with a line size of one word will tend to hit with 0.75 probability in our cache. For instance, the instructions that miss in `tomcatv` have a miss ratio of 0.5. The predictably of these types of instructions can be improved by loop unrolling by the cache line size parameter. Regardless, instructions that account for a small fraction of dynamic references are responsible for a large fraction of the overall misses and have a hit ratio significantly less than the global hit ratio.

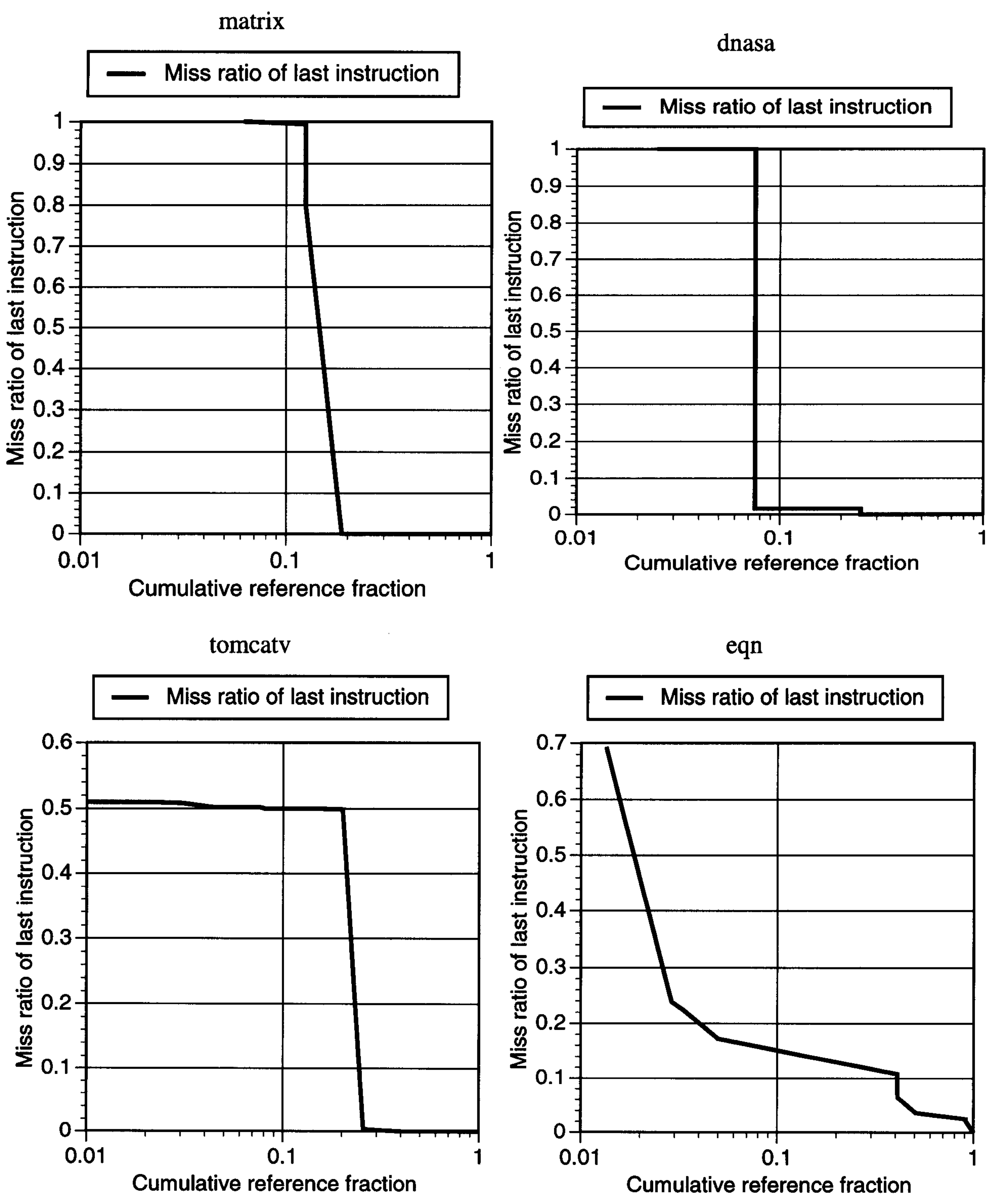

Figure 3.Miss ratio of last load instruction versus cumulative reference fraction
(loads sorted by decreasing miss ratio)

5.5 Performance comparisons after static annotation

Since distinct annotations for each reference in the trace is not feasible, we investigate the performance improvements possible through a static annotation of instructions. In a static annotation, each static load is annotated with 'was replaced' or 'was retained'. A prefetch/pretouch is associated with all dynamic references generated by a load annotated with 'was replaced'. An unprefetched or residual miss is associated with each dynamic reference generated by a load annotated with 'was retained' that misses the cache. Control and data dependencies may constrain the scheduling freedom associated with a static load annotated with 'was replaced'. Consequently, it may not be possible to schedule prefetches/pretouches sufficiently in advance to hide the entire cache miss latency. We do not address this issue in this subsection and we assume that all loads annotated with 'was replaced' are prefetched.

Load/store profiling information is obtained through cache simulations and is used to annotate the static instructions in the program based on a threshold. All instructions that have a miss ratio greater than the threshold are labeled as long latency or prefetch/pretouch instructions and the rest are labeled short latency. This scheme can be successful if it reduces the miss ratio sufficiently without greatly increasing the number of prefetches. We characterize these two factors by two metrics representing the benefit and cost of prefetching: *residual miss ratio* which is the ratio of unprefetched cache misses over total references and *prefetch ratio* which is the ratio of prefetches to the total references in the original unannotated program.

In Figure 4, we plot these two metrics versus threshold for the four benchmark programs. The shape of all four curves are similar; when the threshold is reduced below a critical value there is an abrupt reduction in residual miss ratio and a corresponding increase in prefetch ratio. However, note that the y-axis scales for prefetch ratio and residual miss ratio are different for each benchmark. Thus, in the case of `matrix`, the scales for prefetch ratio and residual miss ratio are identical and for a suitable threshold the residual miss ratio falls to zero even though the prefetch ratio does not exceed the initial miss ratio. Thus, the residual miss ratio can be reduced to zero without incurring any unnecessary prefetches (prefetches that hit in the cache). The behavior of `dnasa` is also close to ideal. For a threshold below 0.8, the residual miss ratio drops to 0.004 and the prefetch ratio does not exceed the initial miss ratio. In contrast, `eqn` has a prefetch scale extending to 0.5 while the residual miss ratio scale extends only to 0.07. Only at a comparatively low threshold does the residual miss ratio does drop significantly below the initial miss ratio; even then the prefetch ratio increases to more than 0.4. Thus, in `eqn` and to a lesser extent in `tomcatv`, the residual miss ratio drops significantly only when the threshold is small and the prefetch ratio is high.

Overall, the static annotation scheme has the potential for reducing miss ratios appreciably at the expense of issuing only a small number of prefetches relative to

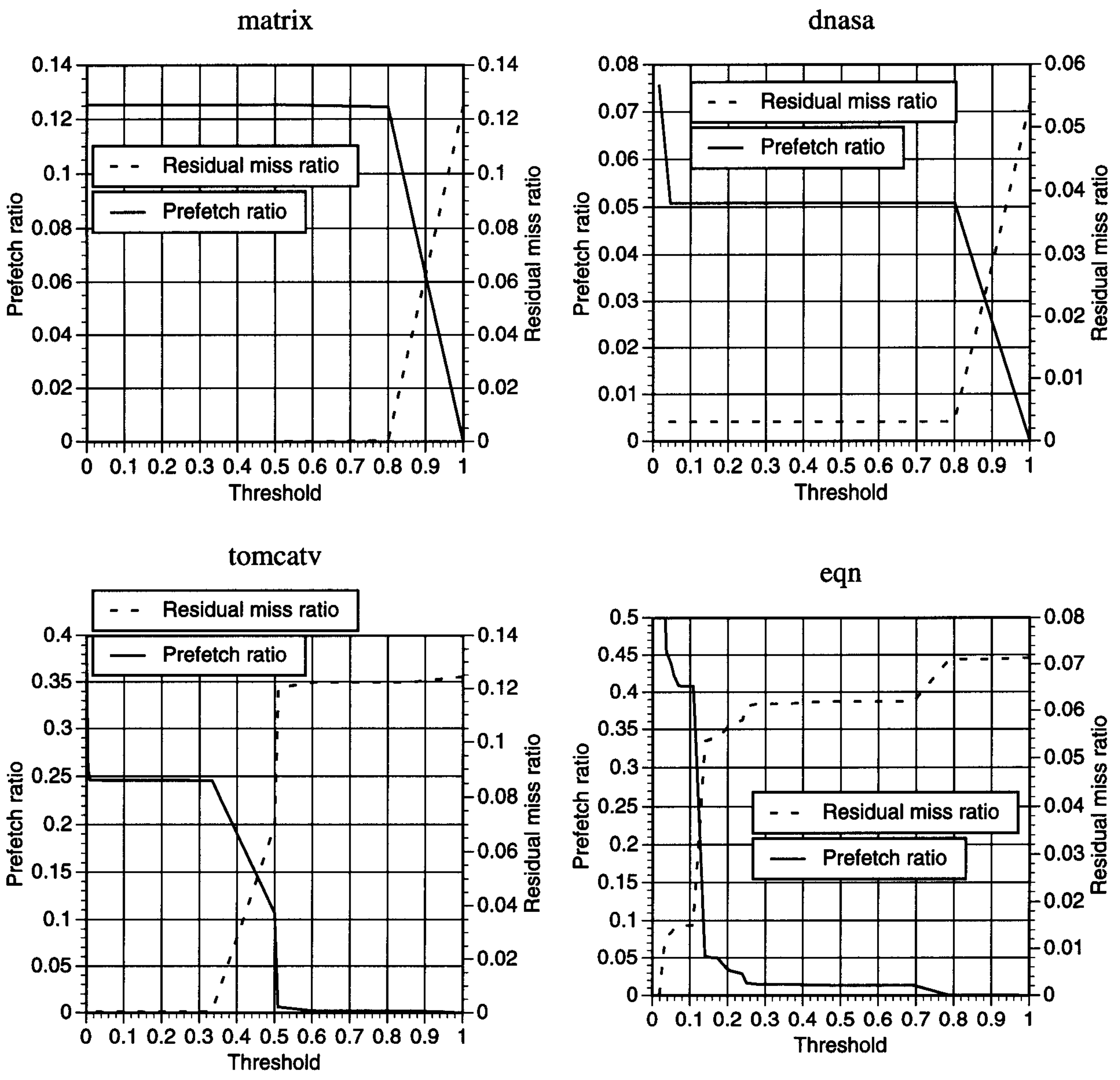

Figure 4.Effectiveness of prefetch annotations

total references. As discussed earlier, in order to prefetch successfully, we also need to move a load up in the schedule so that the distance between the long latency load (or prefetch) and its use covers the miss penalty. In general, control and data dependencies will limit the distance by which a load can be moved ahead of its use. But, since the experiments indicate that the prefetch ratio is usually low, the fraction of loads that have to be scheduled as misses is also low.

The choice of a threshold should be governed by a tradeoff between the cost of prefetches and the benefit arising from a lower miss ratio [25]. Lower residual miss ratios reduce overall cache miss stalls. The cost of prefetches is a function of both the number of prefetches and the amount of instruction level parallelism in the program. In a program with low instruction level parallelism, unnecessary prefetches constrain scheduling decisions and increase critical path lengths. On the other hand, in a program such as `matrix`, the instruction level parallelism is very large and a large prefetch ratio does not affect the schedule length significantly. Therefore, a low threshold is desirable because it can potentially reduce the residual miss ratio and the costs due to prefetches are not significant. In this subsection, we presented the costs and benefits as two separate measures; in the next subsection, we combine the costs and benefits into a single performance measure.

5.6 Characterizing net benefits of profile-driven prefetching

In this subsection, we propose the effective data cache miss metric to characterize the performance impact of caches in systems with lock-up free caches and/or executing selective load-scheduled programs. Under a simple machine model, this metric can be evaluated using the cache hit-miss data obtained through our cache simulation. The results demonstrate that the effective miss ratio can be reduced considerably using cache profiling over a fairly large parameter space.

Consider systems in which a cache miss stalls the entire processor till the data is available and in which all loads are scheduled using hit latencies. For such systems, the cache miss ratio is a simple and effective metric for characterizing the impact of data caches. The cache miss ratio metric can be easily obtained through trace-driven simulation. The performance impact due to data cache misses is proportional to the cache miss ratio, since the stall cycles introduced by the data cache is the product of the cache miss ratio, m, cache miss penalty, l, and the load fraction, f. However, this metric is not as accurate in a modern system with a lock-up free cache where cache misses do not necessarily stall the processor. Even a count of stall cycles may not accurately measure the impact of caches, because the compiler may have inserted prefetches to avoid stall cycles but the performance impact of prefetch-associated operations may not be accounted by stall cycle counts. In selective load scheduled lock-up free systems, a more sophisticated measure is required to estimate the impact of caches. A useful measure should reflect the actual performance impact of caches while being as simple to obtain as possible. Before introducing the effective miss ratio measure, we introduce the data cache Cycles Per Operation (CPO) measure because the link between data cache CPO and performance is clearer.

Let $E(C_x, C_y)$ be the execution time of a program compiled for a system with a cache, C_y and executed on a system with a cache C_x. Let C_∞ represent a perfect cache that never misses. The data cache Cycles Per Operation (CPO) of a program with O useful operations is

$$\begin{aligned} DataCacheCPO &= (E(C, C) - E(C_\infty, C_\infty))/O \\ &= ((E(C, C) - E(C_\infty, C)) + (E(C_\infty, C) - E(C_\infty, C_\infty)))/O \end{aligned} \tag{1}$$

The first line of the above equation defines *DataCacheCPO* as the difference between $E(C, C)$, the execution time of a program compiled for a finite size cache, C and executed on the same cache, C, and $E(C_\infty, C_\infty)$, the execution time of the same program when compiled for and executed on a perfect cache, C_∞. Since a perfect cache never misses, the schedule for the second case, $E(C_\infty, C_\infty)$, corresponds to an all-hit schedule. The second line of the above equation shows that the *DataCacheCPO* can be subdivided into two components: $E(C, C) - E(C_\infty, C)$ the increase in execution time of a program scheduled for a cache, C, when executed on a finite cache, C, as opposed to a perfect cache and $E(C_\infty, C) - E(C_\infty, C_\infty)$ the increase in execution time between compiling for a finite cache, C, over a perfect cache, C_∞, when both schedules are executed on a perfect cache, C. The first component contains stall cycles introduced by executing on a finite cache, C, and the second component contains increases in schedule length due to compiling for a finite cache, C. The effective data cache miss ratio, *EffMissRatio*, is defined as

$$EffMissRatio = DataCacheCPO/(f \cdot l) \tag{2}$$

where f, the load fraction, is the ratio of useful load operations to total useful operations in the program and l, the cache miss latency, is the miss penalty of missing in cache, C. For a conventional system, where the processor stalls on each cache miss and where all loads are scheduled with the cache hit latency, *DataCacheCPO* is the product, $f \cdot m \cdot l$ and *EffMissRatio* is just m, the conventional data cache miss ratio.

Selective load scheduling can improve *DataCacheCPO* and consequently *EffMissRatio* by reducing the first component in equation (1) (stall cycles) but at the expense of increasing the second component in (1) (schedule length). Thus, in contrast to all-hit scheduling, profile-driven annotation and scheduling of loads as long latency has certain benefits and costs. The benefits can be characterized clearly in a statically-scheduled latency-stalled machine. In such a machine, loads are scheduled either with the hit latency or the miss latency and the processor stalls for miss-penalty cycles if a short latency load misses. The processor does not stall on loads that are annotated and scheduled as long latency. Therefore, if the cache miss penalty is l cycles, the benefit of scheduling a load as a long latency is l cycles each dynamic instance that the load would have otherwise missed the cache. In a dynamically-scheduled machine, the benefits cannot be characterized as easily. The stall cycles may be reduced but the reduction in stalls may range between zero and l cycles for each missing load. Furthermore, the processor may not entirely stall but the sustained issue rate may be reduced. In contrast to the benefits, the costs are even more difficult to characterize, since they are a function of the actual schedule generated. In a highly parallel application, there is likely to be sufficient flexibility to schedule a load with the miss latency without lengthening the critical path and hence the schedule length. Aspects of the compiler such as the size of scheduling regions and of the architecture such as the number of registers available, influence the increase in schedule length. The costs associated with scheduling a load with miss latency is a function of the application, compiler, and architecture and is distributed over many parts of the system, such as increase in schedule length, increased register pressure, and cache effects.

Regardless of the difficulty in precisely identifying costs and benefits, it is important to form an initial estimate of the likely benefit of cache profiling. Since the compiler infrastructure required to selectively schedule loads using cache profile information is not yet in place, we form an initial estimate of the potential usefulness of cache profiling using cache simulation data as follows. Since the characterization of the costs is not within the scope of this report, we lump all these costs in a single parameter which we call *schedule length expansion, s*. The value of s ranges from 0 through l, the miss latency and is typically smaller for highly parallel programs, compilers with large scheduling regions and architectures with a large number of registers. This cost is incurred on each dynamic execution of a load. Since we do not have additional information about the cost associated with individual loads, we will assume that all loads have a uniform cost of s cycles per dynamic execution. In practice, the cost may be lower for loads in highly parallel regions of the program and higher in other regions. Furthermore, we assume that every short latency load that misses the cache stalls the processor for l cycles. This assumption is consistent with a latency-stalled processor in which the handling of short latency misses is not overlapped.

In the following discussion, we will consider a code scheduled with hit latencies running on a perfect cache (which always hits) as the base case. Consider a load with a miss ratio of m_i. This load can either be predicted to hit or predicted to miss and in each case scheduled accordingly. If the load is hit-scheduled, then a cost of l cycles is incurred on each cache miss. The net increase in cycle length over the base case is $m_i \cdot l$. On the other hand, if the load is miss-scheduled, then a cost of s cycles is incurred on each dynamic executionof the load. The net increase in cycle length is s cycles. Therefore, it is beneficial to schedule a load with the miss latency only if $m_i \cdot l > s$ or equivalently $m_i > s/l$. In order to verify this statement, consider each of the two extremes (i.e. $s = 0$ and $s = 1$). In the first case, the scheduling cost of a long latency load is zero, as for instance inside a software pipelined loop with no register pressure. In such cases, where there is sufficient parallelism, the above statement says that all loads should be scheduled as long latency because s/l is zero. At the opposite extreme, when $s = l$, the expression states that all loads should be scheduled as hits. When the computation is dependence constrained and any increase in latency for a load results in an equivalent increase in schedule length, there is no benefit in miss-scheduling a load. This statement agrees with intuition at the two extremes.

In order to characterize the net benefit of profile-driven load scheduling, we consider the following schedules and executions of a program.

1. Schedule with all loads hit-scheduled running on a perfect cache. The execution time of this schedule, $E(C_\infty, C_\infty)$, is the execution time of the program in the absence of data cache effects.
2. Schedule with dynamically annotated loads, so that the long latency scheduling cost of s cycles is incurred only on dynamic loads that actually miss. The execution time of this program represents the best achievable for a non-zero scheduling cost, s. This schedule represents an ideal usually unachievable bound.
3. Schedule with all loads scheduled as hits. The execution time, $E(C, C_\infty)$, represents current practice because most RISC compilers assume cache hit latencies for all references.
4. Schedule with all loads miss-scheduled. This represents a feasible strategy on software pipelinable loops.
5. Profile-driven scheduling of loads. All loads with a miss ratio greater than s/l are scheduled as misses, rest as hits.

Now, we compare the execution time of the above schedules for a range of values of s. The ratio s/l can range from 0 through l, where 0 corresponds to highly parallel, latency insensitive code and 1 to highly sequential, latency tolerant code. We will use the difference between the CPO (Cycles Per Operation) of a schedule and the CPO of the base Schedule 1 as a measure of data cache effects on execution time. As shown in Figure 5, the CPO contribution due to data cache effects for the perfectly annotated Schedule 2 is $f \cdot m \cdot s$, because a dynamic load reference is annotated only if it misses and every such load reference incurs a scheduling cost of s. For Schedule 3 with all loads scheduled as hits, the CPO contribution is $f \cdot m \cdot l$, because every load that misses incurs a cost of l cycles. For Schedule 4 with all loads scheduled as misses, the CPO contribution is $f \cdot s$, because every load incurs a cost of s cycles. Finally, the CPO contribution of Schedule 5 depends on the fraction of loads scheduled with miss latency (having a miss ratio greater than s/l) and the fraction of loads scheduled with hit latency (those with a miss rate less than s/l) that miss. Representing these two fractions by p_m and p_{hm} respectively, the CPO contribution is $f(p_m \cdot s + p_{hm} \cdot l)$. For a given program scheduled using profile information from one input data set and subsequently running on the same data set, the CPO contribution of Schedule 5 will always be upper bounded by Schedules 3 and 4.

Figure 5 illustrates that at the two extremes, the current practice of either scheduling all loads as cache hits or as cache misses achieves the ideal bound and thus no additional improvement is available through profiling. If the cost of scheduling loads with miss latency is zero, as may be the case when the program is highly parallel,

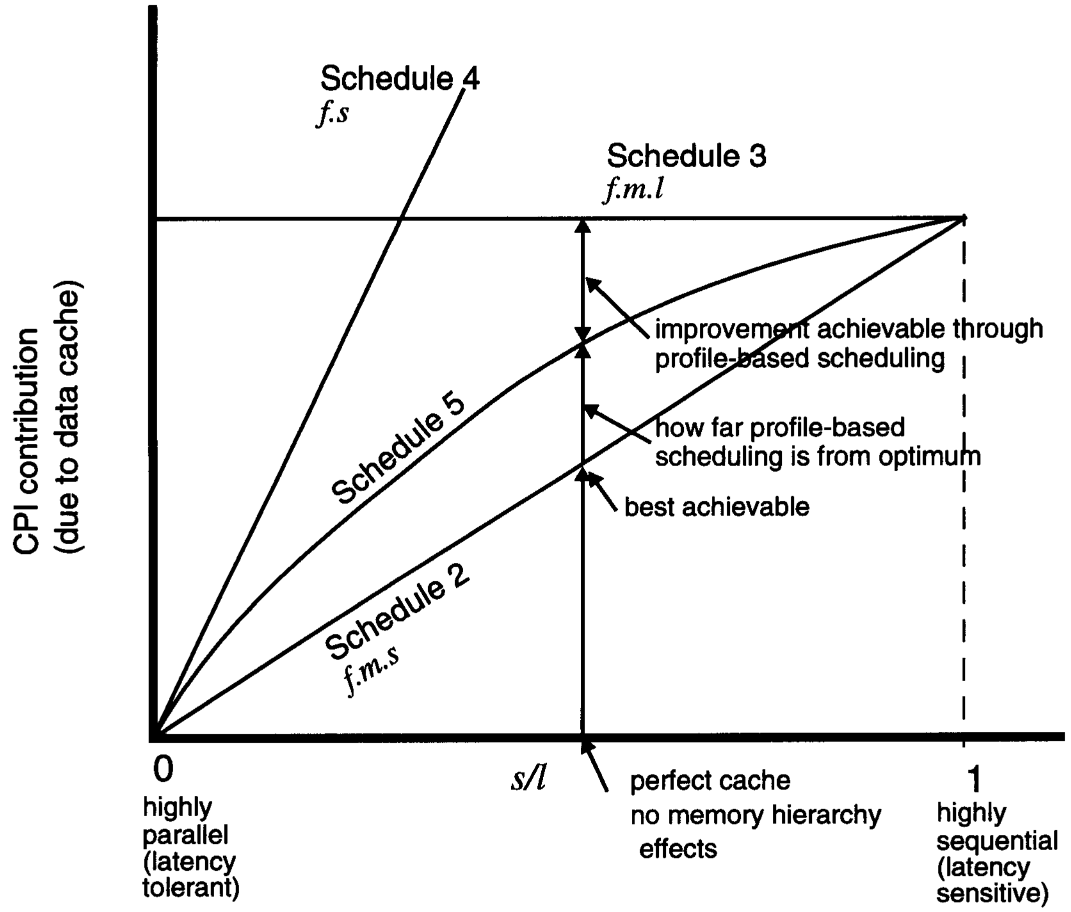

Figure 5.CPI contribution due to data cache effects
(curves are illustrative and not based on actual data)

Schedules 2, 4, and 5 all schedule all loads with the miss latency. The CPO contribution due to data cache effects is zero because all loads are scheduled with miss latencies. Note that even if the program is highly parallel, the cost associated with scheduling loads as long latency may not be zero because of the resource requirements of long latency loads such as additional register pressure At the other extreme, if the scheduling cost is *l* as may be the case when the program is highly sequential, Schedules 2, 3, and 5 will all schedule all loads as hits.

The traditional measure of cache performance is the miss ratio, the fraction of accesses that miss the cache and are satisfied from the next level of the memory hierarchy. This measure is inadequate for a machine with a lockup-free cache, which supports non-blocking requests. An optimized program may have a relatively large cache miss ratio but by suitable prefetching the processor may hardly be stalled by the cache. In this case, the cache miss ratio indicates a large performance degradation due to the cache, even though the cache has little overall performance impact. However, if the scheduling of the prefetches was achieved at the expense of stall cycles or additional operations, the overall performance is reduced relative to a perfect cache system where such scheduling was unnecessary. In the following experimental results, we present the *effective cache miss ratio* which is the CPO contribution scaled down by the product of the miss latency, l, and the fraction of load references, f, as indicated in equation (2). Thus, for Schedule 3, the effective miss ratio is $(f \cdot m \cdot l)\ (f \cdot l) = m$, the traditional miss ratio. But, for other schedules, the effective miss ratio accounts for both the misses on hit-scheduled loads and the costs of miss-scheduling loads.

In Figure 6, we show the effective miss ratio of four programs for the four schedules described earlier. The profile driven behavior of `matrix` is very close to the ideal. The programs `dnasa` and to a lesser extent `tomcatv` also have very good profile behavior. But, `eqn` does not perform much better than the traditional all-miss scheduling or all-hit scheduling. In the absence of results from a selective load scheduler, we cannot comment on the precise range of values of s. But, for `matrix` and `eqn`, regardless of the value of s, the performance of the profile based curve is close to ideal. Thus, for these two programs, profile based optimization is likely to reduce the impact of data caches to the minimum regardless of the precise value of s.

5.7 Relationships between results

In Subsections 5.3 and 5.4, our emphasis was on presenting the caching behavior of loads. Subsection 5.3 demonstrated that few loads account for most cache misses and Subsection 5.4 showed that loads generally tend to have a high miss ratio or a miss ratio close to zero. In Subsections 5.5 and 5.6, our emphasis was on the potential costs and benefits associated with attempting to exploit these properties. In Subsection 5.5, we presented two measures: the prefetch ratio estimating the cost and the residual miss ratio estimating the potential benefit. In Subsection 5.6, we combined the costs and benefits into a single performance measure, the effective miss ratio.

In this subsection, we demonstrate that there is a strong interrelationship between the properties presented in Subsection 5.4 and the performance estimates obtained in Subsections 5.5 and 5.6. Figure 7 shows an illustrative plot of the miss ratio of last instruction versus cumulative reference fraction. As in Subsection 5.4, we assume that the instructions are sorted by decreasing miss ratio. For each subset of loads taken from the head of this sorted list, we plot the miss ratio of the last load versus the cumulative reference fraction of all loads in the subset. Unlike Subsection 5.4, we assume that the axes of the plots are linear. The performance results in Subsection 5.5 can be derived from this plot as follows. For each threshold value, locate the corresponding point on the y-axis. Draw a horizontal line to the curve and drop a vertical line to the x-axis. The intercept on the x-axis is the prefetch ratio because all

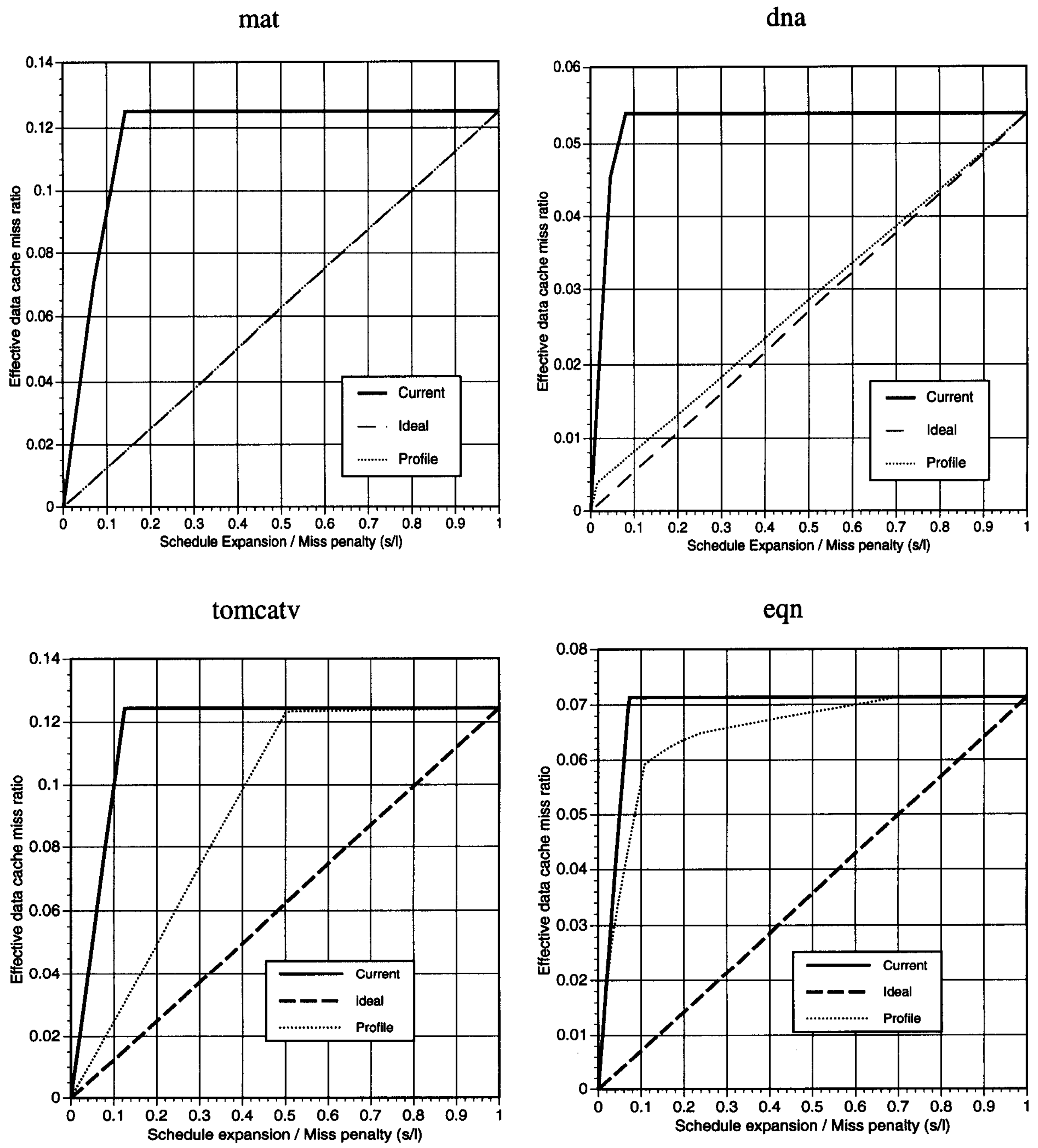

Figure 6.Effective miss ratio vs. scheduling cost

loads to the left of the intercept point are prefetched. All loads to the right are not prefetched and any misses from these loads contribute to the residual miss ratio. Therefore, the residual miss ratio is the area under the curve beyond the x-axis intercept. Thus, the performance results in Subsection 5.5 which plot prefetch ratio and residual miss ratio can be derived directly from the plots in Subsection 5.4. Similarly, the performance results in Subsection 5.6 can be derived from the results in Subsection 5.4 as follows. In this case, use *s/l* to obtain the x-axis intercept as before. Since all loads to the left are prefetched at a cost of *s* and since the loads to the right incur cache misses, the effective miss ratio is obtained by summing the area denoted by residual miss ratio and the rectangular region denoted by Prefetch Ratio * s/l in the figure.

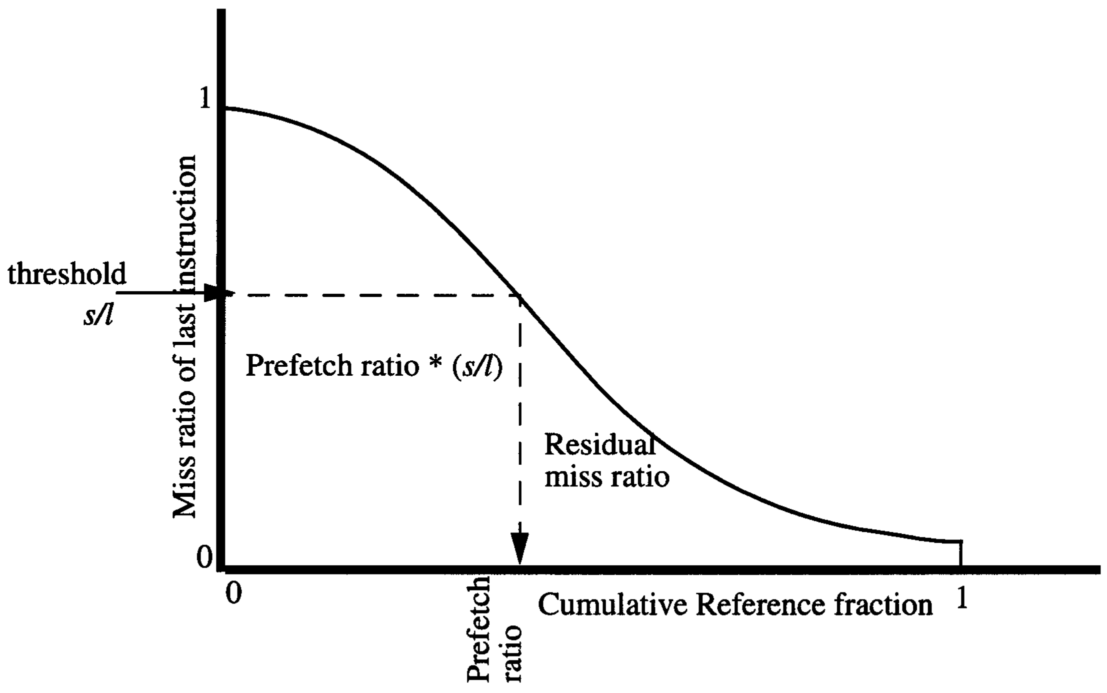

Figure 7.Derivation of performance curves from property curve

This relationship shows that the potential performance benefits of cache profiling stem from the dichotomous caching behavior of loads, i.e., some loads have high miss ratios and others very low miss ratio. For instance, if all loads had the same miss ratio, the curve in Figure 7 would be a straight horizontal line. If the threshold is less than the global miss ratio, no loads are prefetched; otherwise, all loads are prefetched. In one case, the labeling of loads corresponds to all-hit scheduling; in the other case, to all-miss scheduling. Thus, profiling offers no advantage when all loads behave identically.

6 Effect of different data sets

Optimizations based on profile information from one input set may lead to poor performance for some other input sets. In order to evaluate the viability of cache profiling, we investigate the effect of using distinct data sets for profiling and executing the program. In the previous section, we presented several performance measures when the same program is used for profiling as for running. In this section, we present similar performance measures using various input sets for profiling and a single input set for rerunning the program. We refer to the various input data sets used for profiling as the training data set (TDS) and the input set used for re-executing the program as the running data set (RDS).

6.1 Methodology

An earlier study by Yeh and Patt on branch profiling collected several data sets for most of the SPEC 89 benchmarks. Our study uses a subset of this collection of data sets. Table 5 gives a list of SPEC benchmarks and their datasets that we used. The other SPEC programs such as `tomcatv`, `nasa` and `matrix` do not read a dataset. Each of the SPEC89 programs was simulated on each of the available data sets for that benchmark. Recall that the performance figures presented in the earlier section firstly order all the loads (e.g. decreasing cache miss ratio) and then determine performance statistics. In contrast to the previous section, we use the cache simulation results on one data set (training data set) to order loads and the results on another data set (running data set) for determining performance statistics. Thus, the number of cache simulation runs required for the results presented here on a benchmark with n data sets is only n not n^2. Even though we obtained results on all the benchmarks listed in Table 5, we only present the results for `gcc`, `li`, `espresso`, and `spice`. Since we do not present results on the benchmarks `doduc` and `eqntott` in this section, we do not identify the datasets used for running and training. For the four benchmarks for which we present results, Table 5 lists the keyword RDS followed by a parenthesized list of data set(s) that we used as the running data set and similarly, lists the keyword TDS followed by a parenthesized list of data set(s) that we used as the training data set. For `gcc`, we used two training data sets identified by TDS 1 and TDS 2 in Table 5 and in subsequent figures. The annotation 'ref' associated with a data set indicates that this data set is the reference data set used for SPEC benchmarking. The simulation environment is otherwise identical to that described in Section 5.1. The caches were simulated for 100 million references.

6.2 Distribution of references and misses

The analysis and results presented in this subsection are similar to those in Section 5.3. The graphs in Figure 8 plot the fraction of dynamic references made by a certain number of load instructions when a benchmark is executed on the running data set. The x-axis is on a log scale in order to magnify the left hand region showing

Table 5. Datasets used for benchmarks

Programs	Datasets
spice	RDS (greybig[ref]), TDS (grey short)
doduc	tiny, small, large[ref]
espresso	RDS (bca, esp [ref], ti), TDS (cps, tial)
eqntott	ex0, ref
li	RDS (div2, li-input[ref], sort), TDS (hanoi, power)
gcc	RDS (cexp, dbxout, emit-rtl), TDS 1(explow, gcc, genrecog), TDS 2(jump, recog, toplev)

the behavior of instructions with high reference coverage. For gcc, approximately 3000 out of 25395 loads account for 90% of the dynamic references. In addition, for each training data set, the load instructions are sorted by their individual miss ratios on that training set. Then, the cumulative miss fraction in the running data set accounted by a certain number of load instructions is plotted, with one curve for each of the training data sets. As the graph for gcc and li indicates, a certain number of loads account for a much larger fraction of the misses than references, even when the training data set is different from the running data set. However, the training data sets are not good predictors of which loads have large miss fractions, in the other two programs. In spice, just four loads account for half the misses and nine instructions account for 90% of the misses. However, the twelve loads with the highest miss fractions in the training data set account for hardly any misses in the running data set. In the case of spice, this behavior is explained by the fact that the training data set exercises a considerably different part of the program than the running data set.

6.3 Distribution of dynamic trace annotations

Figure 9 is similar to Figure 3 in Section 5.4. For each training data set, we sort the instructions by decreasing miss ratio on the training data set. We divide the list into two segments, the segment containing the head of the list having high miss ratio and the tail segment having low miss ratios. We plot the average miss ratio of the head segment versus the reference fraction of this segment on the running data set. Similarly we plot the hit ratio of the tail segment versus the reference fraction of the head segment. Note that in contrast to Section 5.4 where we plotted the miss ratio of the last instruction, in this subsection we plot the average miss ratio. If we plot the miss ratio of the last instruction when profiling and running on different data sets, the plot will tend to be highly irregular and non-monotonic. The average miss ratio plot is

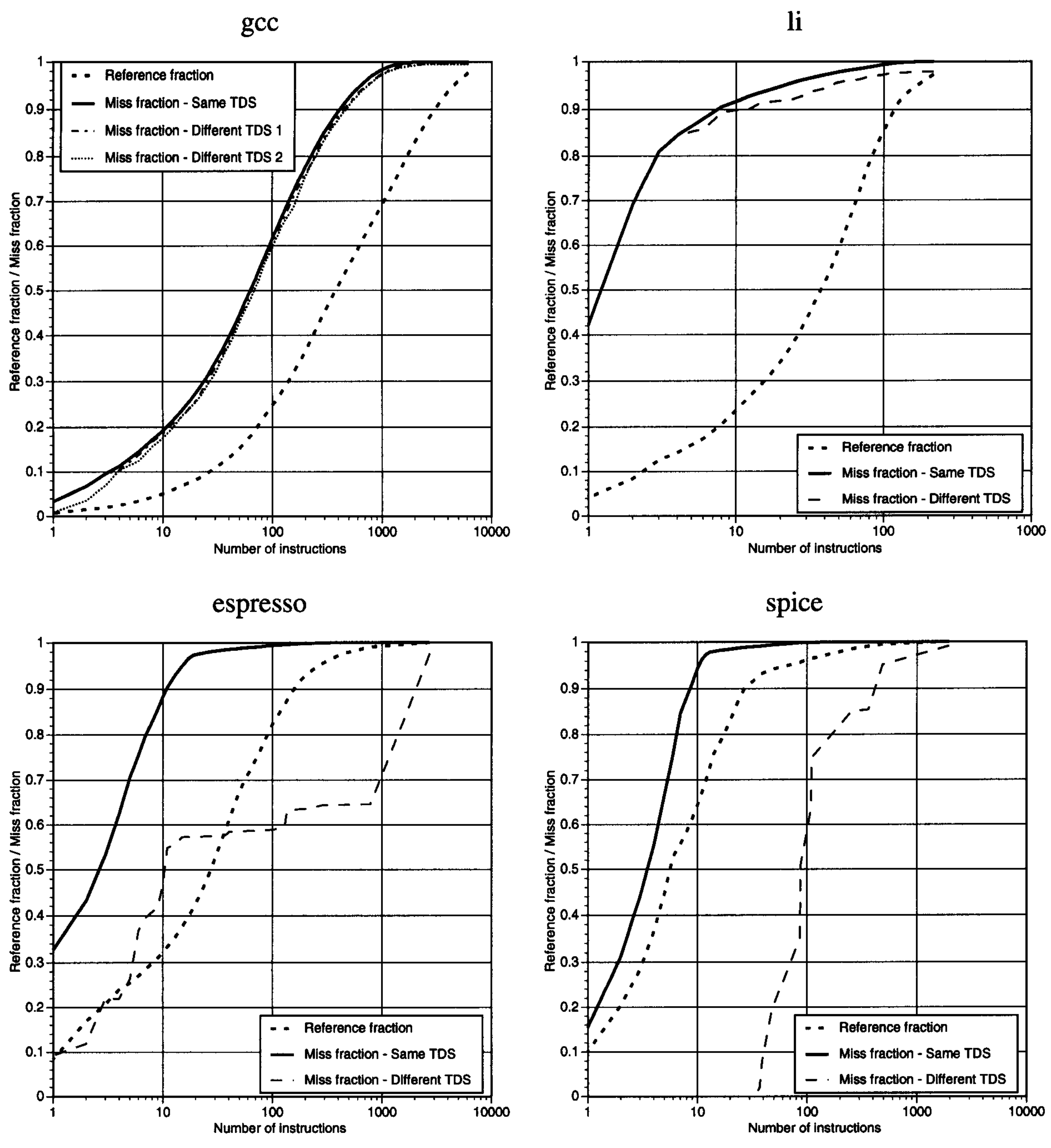

Figure 8.Cumulative reference and miss fraction

much smoother. In the ideal case, where some loads identified through profiling on the training data set always misses the cache in the running data set and other loads always hit the cache, the two curves are almost step functions. The average miss ratio curve starts at one, and then exponentially decays to the global miss ratio. The average hit ratio starts at one on the right-side and then decays to the global hit ratio on the left-axis. The plot for `gcc` and `li` exhibits near ideal behavior. In the case of espresso and especially spice, the cumulative miss ratio curve is quite different when the training is performed on a different data set.

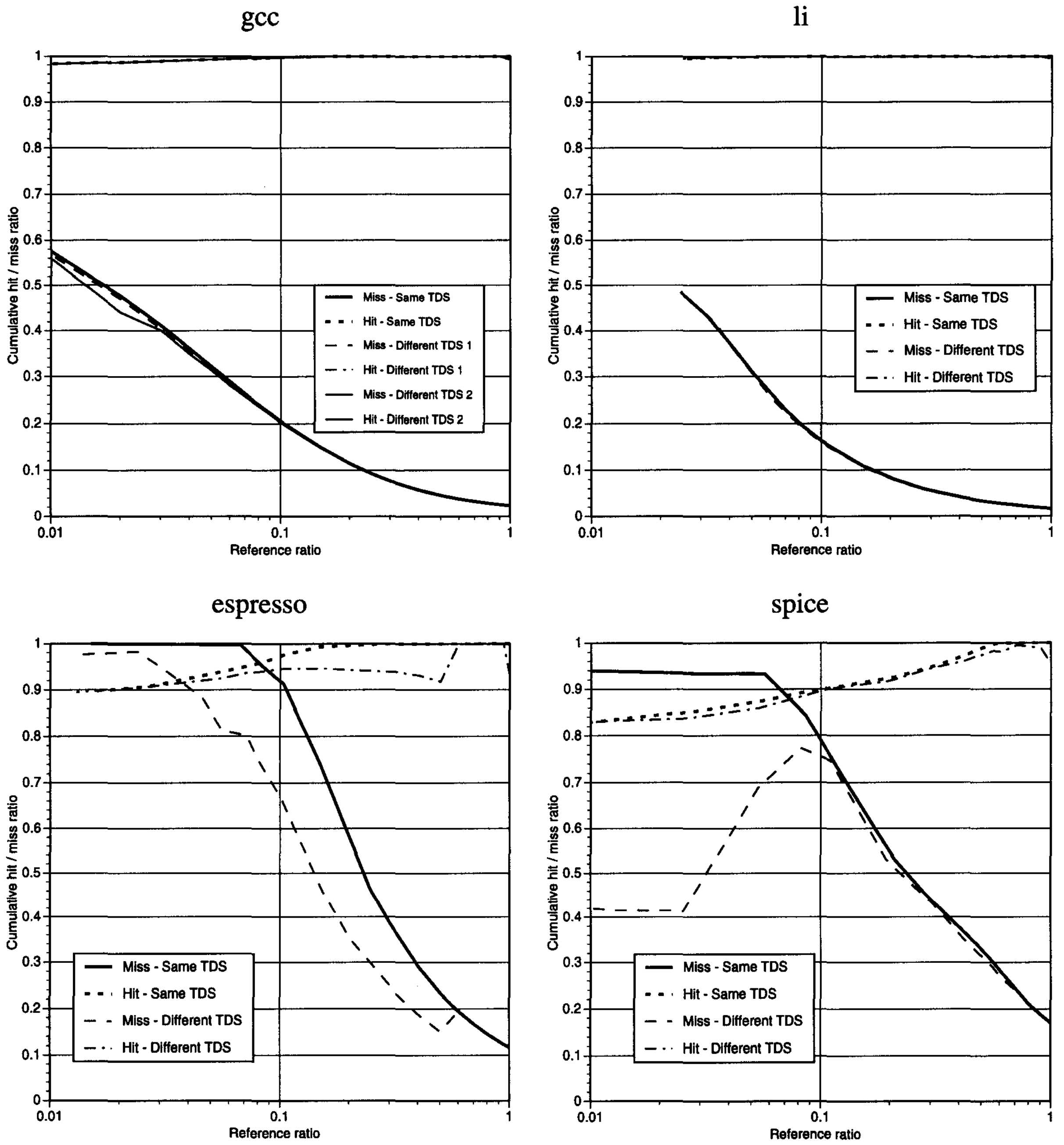

Figure 9.Cumulative hit/miss ratios for multiple data sets

6.4 Performance comparisons after static annotation

In this subsection, we present results similar to Section 5.5, but on benchmarks with multiple data sets. In Figure 10, we plot the residual miss ratio and the prefetch ratio versus threshold for the four benchmark programs. In `gcc`, the residual miss ratio decreases gradually as the threshold is decreased, with an accompanying increase in the prefetch ratio. In `li`, there is steep decrease in the residual miss ratio at a threshold of 0.4. In both cases, the behavior when the profiling is done with a different training data set is not significantly worse. In contrast, for `espresso` and `spice`, the residual miss ratio curve when profiling with a different data set is significantly higher than when profiling with the same data set.

6.5 Modeling net benefit of cache profiling.

In this subsection, we present results similar to Section 5.6. As before, we plot the effective miss ratio when ideal dynamic annotations are used (Ideal) and when current all-hit or all-miss scheduling is employed (Current). For each training data set, the miss ratio of each instruction is determined through cache simulation. Depending on whether the miss ratio of an instruction is greater than s/l, each load is tagged as scheduled with hit or miss latency. We plot the effective miss ratio using cache profiling for the full range of values of s/l, presenting one curve for each training data set. For `gcc` and `li`, the performance when profiling is done with a different data set approaches the performance achieved through the same data set. Thus, `gcc` and `li` are relatively insensitive to the two training data sets that we used with respect to the performance benefit of cache profiling. In the case of the other two programs, `espresso` and `spice`, the performance when a different training data set is used is significantly worse than when the same data set is used both for training and running. Under self-profiling, the effective miss ratio is guaranteed to never exceed the Current curve, but there are no such bounds when a different data set is used for training. In both espresso and spice, the effective miss rate under different TDS profiling is more than all-miss scheduling when s/l is small.

6.6 Correlation between miss ratios on different data sets

In the previous subsection, we evaluated the potential benefit of profiling. In this subsection, we investigate some fundamental properties that determine whether a program's cache behavior is sensitive to a range of data sets. Using the results of cache profiling on one data set to optimize a program for other data sets will be effective only if the miss ratio of loads is similar on a range of data sets. The miss ratios on different data sets must be well-correlated. Additionally, loads that account for a large miss fraction are more important. In the graphs in Figure 11, we represent each load with a miss fraction of at least 0.1% on the running data set with a circle located at the coordinates formed by the miss rate on the running and training data sets respectively. If the cache behavior of loads in the training and running data sets were identical, the plot would contain a series of circles located on a diagonal across the plot. The size of the circle represents the importance of a particular load and we consider the miss fraction of a load as the measure of its importance. Again, the cir-

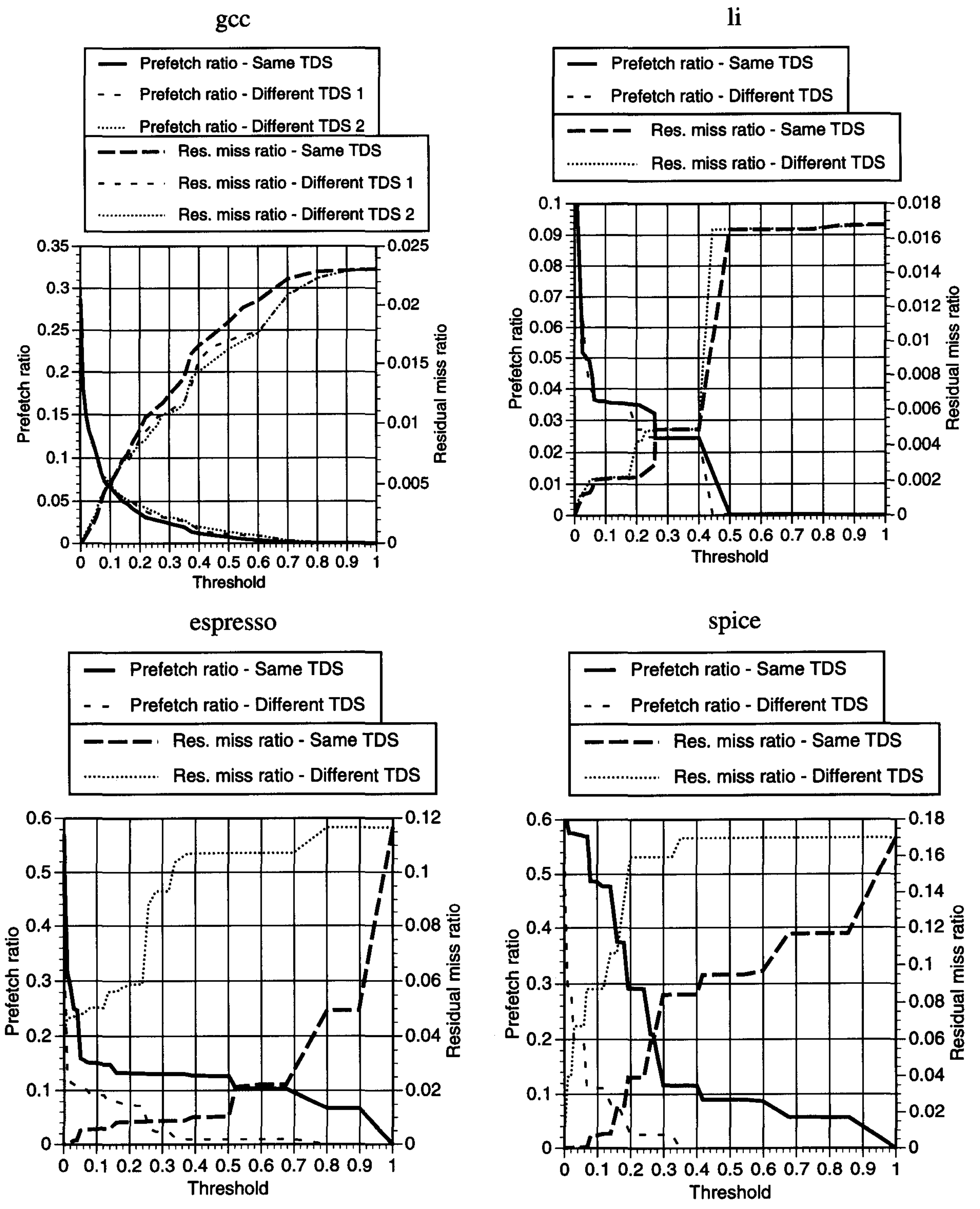

Figure 10.Residual miss ratio and prefetch ratio versus threshold

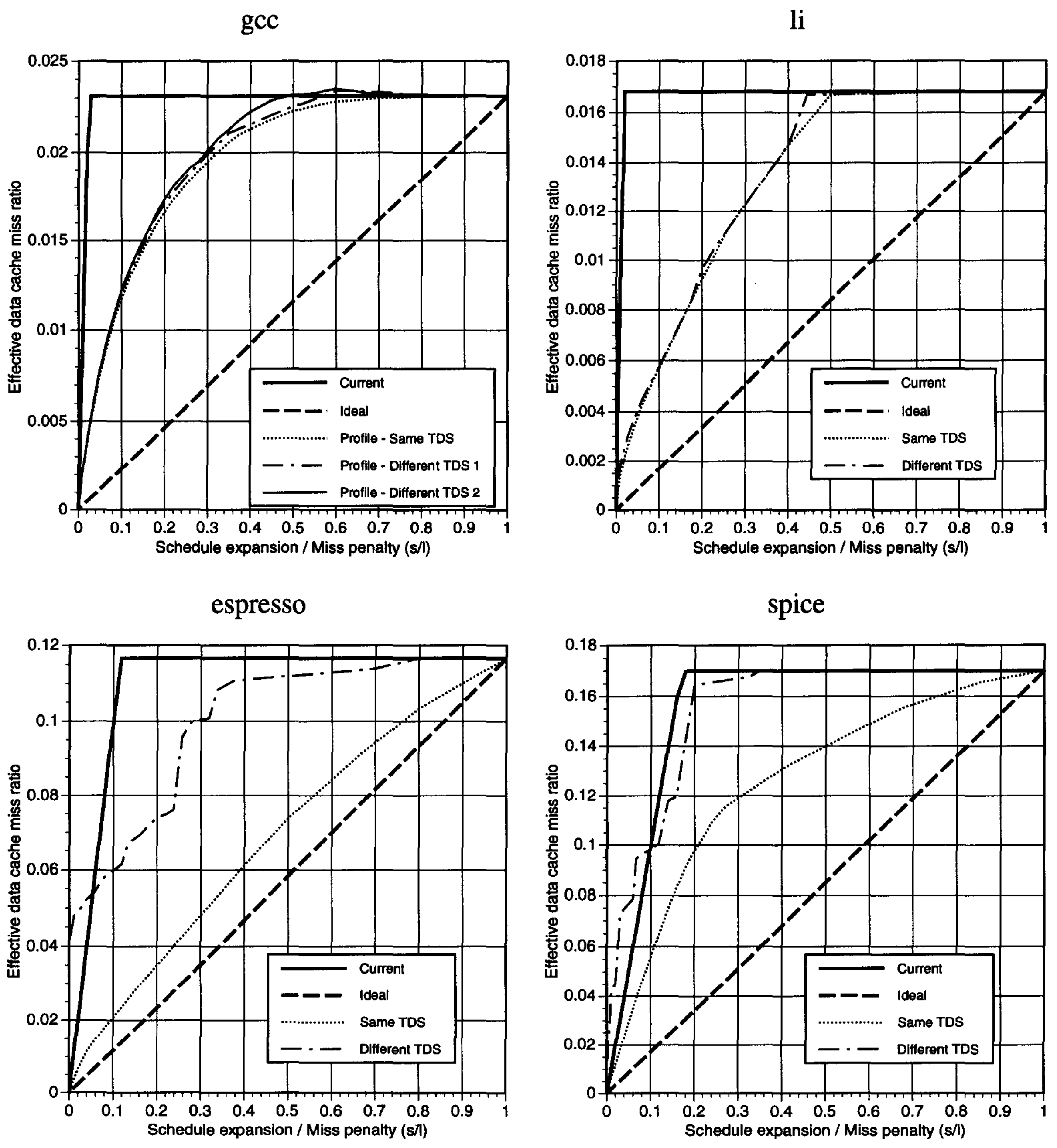

Figure 11.Effective miss ratio for multiple data sets

cles are clustered along the diagonal for gcc and li, but widely distributed for the other two programs. Also, gcc being a large program has many more loads that have a miss fraction of 0.1%, unlike the other programs that appear to be dominated by a few missing loads.

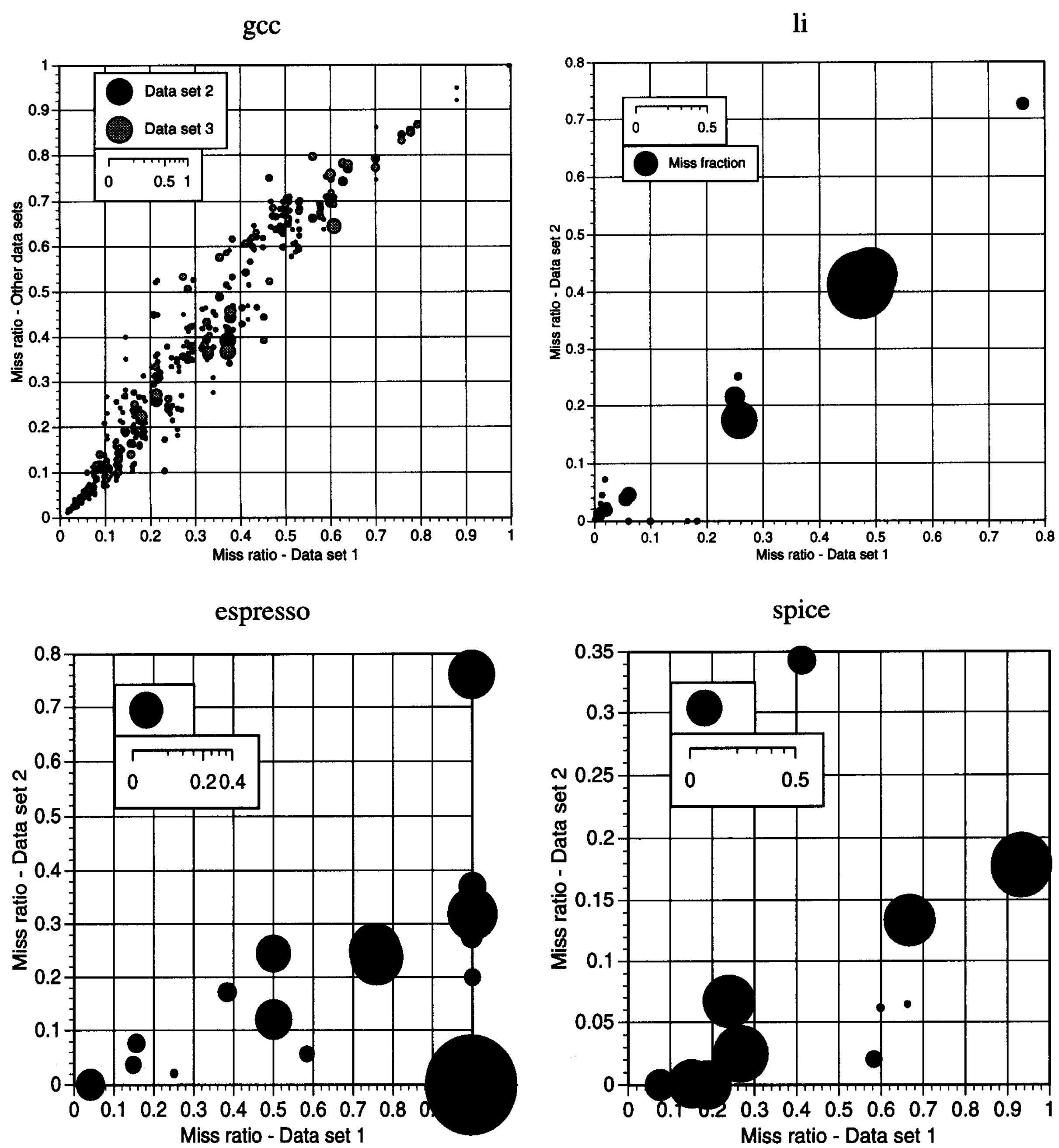

Figure 12.Correlation between miss ratios on different data sets

7 Conclusion and Future Work

Cache control instruction are being implemented in most future systems. However, there has been little attempt at quantifying the gains that these instructions can provide for irregular applications. In this report, we motivate cache control instructions using a blocked matrix multiply example. A system with cache-control instructions is superior to other common architectural alternatives for this example program.

We characterize the misses for several SPEC benchmark programs. For typical first-level cache sizes, the miss ratio is significant when combined with the large and increasing miss penalty, motivating the need for better schemes to reduce cache miss effects on overal performance. We present annotation statistics for the dynamic trace. We demonstrate that hit-instructions accounting for a large fraction of total references almost never miss and that the miss-instructions have a hit ratio less than half the global hit ratio. Under a thresholding scheme that partitions loads into short and long latency, the residual miss ratio due to non-prefetched references can be kept low while not greatly increasing the fraction of references prefetches.

In the future, we plan to integrate the profile-based labeling scheme with a compiler that can use latency-specification to perform better instruction scheduling. A range of scheduling heuristics that employ cache profiling information will be implemented. Our current scheduling framework requires the specification of latencies prior to scheduling. Thus, we initially plan to use a fixed miss threshold to label loads prior to scheduling. In the next phase, the choice of threshold for labeling loads in a scheduling region will be based on an estimate of available parallelism and scheduling flexibility in that region. In the final framework, the labeling decision will be made at the time that each operation is chosen for scheduling. We also plan to account for machine features such as the maximum number of outstanding misses and register constraints.

We plan to improve cache profiling by using some ideas that have been exploited recently in branch profiling. For instance, we plan to implement a path-sensitive cache profiler that collects profiling information for each load in several bins, each bin representing a set of control flow paths. Such profiling is likely to be more accurate and will suggest unrolling or peeling where appropriate. Path-sensitive profiling has improved branch profiling by up to a factor of two and we expect significant improvements in cache profiling. We also plan to identify heuristics that can be employed by a compiler to identify missing references at the source-level. Profiling can also be used for gathering other information regarding the expected behavior of the memory hierarchy. In memory dependence profiling, we determine the dynamic dependencies between stores and later loads. This information can be used by a compiler to move loads ahead of stores, in a system that supports a mechanism to detect and handle cases where the reordered loads aliased into the same location as an earlier store. Memory dependence profiling also provides an useful upper bound on the benefit obtainable using static memory disambiguation algorithms. Reuse profiling determines whether a location accessed by a load/store is subsequently accessed before being replaced from the cache. Cache replacement can be managed using the target cache specifiers described in Section 2 and such reuse profiling information can be used to reduce traffic between levels of the memory hierarchy. Our initial results on using profiling to manage cache contents show a reduction in traffic of less than 10% [23].

The hit-miss behavior of loads is often sensitive to the input data set. Techniques to reduce the sensitivity of cache profiling information to input data sets are required. A straightforward approach is to average the hit-miss statistics over several data sets. In our work, we found that training using two or three input files yielded significantly better results than training with one data set. Also, techniques to weight a load annotation based on the frequency of execution of a load on a particular benchmark may be useful. Currently, a load may be miss-scheduled even if it was executed only once when running on a particular training data set. Further experimentation is necessary to determine the sensitivity of the partitioning of loads into short and long latency to changes in cache configuration. Finally, techniques to reduce the cost of cache profiling are also being investigated.

Acknowledgments

The original idea of cache profiling emerged in discussions between the authors and Michael Schlansker and Rajiv Gupta. Rabin Sugumar implemented the initial set of tools to gather and analyze miss ratios of individual instructions. Jai Menon and Lapyip Tsang conducted the cache simulation experiments and gathered the raw performance data.

References

[1] *Alpha Architecture Handbook – Preliminary Edition*. Digital Equipment Corporation, Maynard, MA, 1992.

[2] V. Kathail, M. S. Schlansker, and B. R. Rau. "HPL PlayDoh architecture specification: Version 1.0." Technical Report HPL-93-80, Hewlett-Packard Laboratories, Feb. 1994.

[3] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. "Predictability of load/store instruction latencies." *Proc. 26th Ann. Int. Symp. Microarchitecture*, pages 139–152, 1993.

[4] D. Callahan and A. Porterfield. "Data cache performance of supercomputer applications." In *Supercomputing '90*, pages 564–572, 1990.

[5] D. Callahan, K. Kennedy, and A. Porterfield. "Software prefetching." In *Proc. of ASPLOS IV*, pages 40–52, 1991.

[6] D. M. McNiven. *Reduction in Main Memory Traffic through the Efficient use of Local Memory*. Ph.D. thesis, University of Illinois, 1988.

[7] A. C. Klaiber and H. M. Levy. "Architecture for software controlled data prefetching." In *Proc. of 18th Intl. Symp. on Computer Architecture*, pages 43–63, 1991.

[8] T. C. Mowry, M. S. Lam, and A. Gupta. "Design and evaluation of a compiler algorithm for prefetching." In *Proc. of ASPLOS V*, pages 62–73, 1992.

[9] T.-F. Chen and J.-L. Baer. "Reducing memory latency via non-blocking and prefetching caches." In *Proc. of ASPLOS V*, pages 51–61, 1992.

[10] W. Y. Chen, S. A. Mahlke, and W. Hwu. "Tolerating first level memory access latency in high-performance systems." In *Intl. Conf. on Parallel Processing*, pages I–36 – I–43, 1992.

[11] J. A. Fisher. "Trace scheduling: A technique for global microcode compaction." *IEEE Trans. on Computers*, C-30(7):478–490, July 1981.

[12] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. "The superblock: An effective technique for flaw and superscalar compilation." *J. of Supercomputing*, 7(1/2):229–248, 1993.

[13] J. A. Fisher and S. M. Freudenberger. "Predicting conditional branch directions from previous runs of a program." *Proc. of ASPLOS*, pages 85–94, October 1992.

[14] A. Krall. "Improving semi-static branch prediction by code replication." *Proc. ACM SIGPLAN Conf. Prog. Lang. Des.& Impl.*, pages 97–106, 1994.

[15] S. McFarling. "Program optimization for instruction caches." In *Proc. of ASPLOS III*, 1989.

[16] W. W. Hwu and P. P. Chang. "Achieving high instruction cache performance with an optimizing compiler." In *Proc. of 16th Intl. Symp. on Computer Architecture*, pages 242–251, 1989.

[17] F. Chow and J. Hennessy. "Register allocation by priority-based coloring." In *Proc. of the 1984 Symp. on Compiler Construction*, pages 222–232, 1984.

[18] P. Chang, S. Mahlke, W. Chen, and W. Hwu. "Profile-guided automatic inline expansion for C programs." *Software-Practice and Experience*, 22(5):349–369, 1992.

[19] M. Martonosi, A. Gupta, and T. Anderson. "Effectiveness of trace sampling for performance debugging tools." In *Proc. ACM SIGMETRICS Conf.*, pages 248–259, 1993.

[20] G. R. Beck, D. W. L. Yen, and T. L. Anderson. "The Cydra 5 mini-supercomputer: architecture and implementation." *The Journal of Supercomputing*, 7(1/2):143–180, 1992.

[21] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. "The Cydra 5 departmental supercomputer: Design philosophies, decisions and trade-offs." *IEEE Computer*, 22(1):12–35, 1989.

[22] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. "A VLIW architecture for a trace scheduling compiler." *IEEE Transactions on Computers*, C-37(8):967–979, 1988.

[23] R. A. Sugumar and S. G. Abraham. "Multi-configuration simulation algorithms for the evaluation of computer architecture designs." Technical Report CSE-TR-173-93, CSE Division, University of Michigan, 1993.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation techniques for storage hierarchies." *IBM Systems Journal*, 9(2):78–117, 1970.

[25] D. R. Kerns and S. J. Eggers. "Balanced scheduling: Instruction scheduling when memory latency is uncertain." In *Proc. of the SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation*, pages 278–289, 1993.